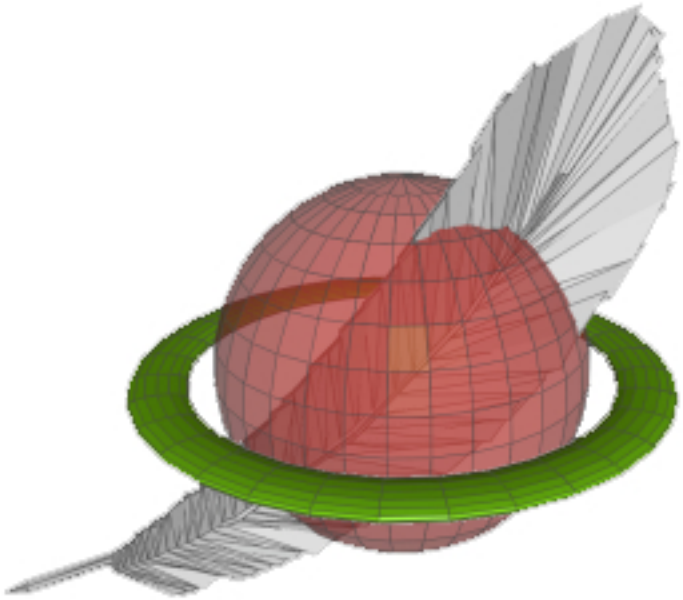
SpatiaLite Cookbook

Author: Alessandro Furieri



2011 January 28

This work is licensed under the Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

using SpatiaLite

a fast and simple practical ***how-to*** for absolute beginners

Introduction / Table of Contents

2011 January 28

Manifest declaration: in Computing there are two ill-fated words: **database** and **SQL**.

Both terms share a long-established and deeply consolidated bad reputation:
simply pronouncing their name will immediately cause strong negative reactions:
"*too much complicated*", "*I'll never be able to understand all this*", "*nerds oddities*" and so on ...

If all this is indisputably true for *plain basic* SQL, what's about the most exotic **Spatial SQL** ?
Obviously, this sounds by far much more complex and intimidating:
for sure only very few specially trained and highly qualified gurus can cope with Spatial SQL, isn't ?

Forget all this, and be open-minded !

(*I was quite going to write*: **Bullshit !** *but I don't feel using abusive words could mark a good style start ...*)

I suppose all the above prejudices simply are the sad consequence of long standing marketing policies.
For many years the DB market was dominated by highly-priced proprietary software:
and the Spatial DB simply represented a narrowest specialized segment (*even more exaggeratedly expensive*) within mainstream DB market.
So the intimidating *sacral aura* surrounding DB technologies was much more a cover-up story to justify abnormally high prices than an objective technical fact.
Happily the actual truth is quite different from this: any normally *computer-skilled* professional people can easily learn and successfully use both SQL and Spatial SQL;
there is absolutely nothing difficult, obscure or much more complex in using them.

And when I say **professional people** I don't necessarily intend *developers*, *computer engineers* or *GIS professionals*:
I'm personally well aware that lots of *ecologists*, *traffic engineers*, *botanicians*, *environmental engineers*, *zoologists*, *public administration officers*, *geologists*, *geographers*, *archeologists* (and many others) successfully use Spatial SQL on their daily activities.

So you can now get a sophisticated, standard and really effective Spatial DB absolutely for free and in the easiest and painless way.
And that's not all: SpatiaLite is strongly integrated into the *open source ecosystem*.
So you can immediately connect any SpatiaLite's DB using e.g. **QGIS** (a well known and widespread desktop GIS app).

You don't believe to my assertions ? Well, try by yourself: come touching with your hands and seeing with your eyes.
What could be better than this approach to get a first-hand neutral, objective and unbiased experience ?

Following this tutorial will approximately take about an hour or two of your precious time:

that's not a too much demanding task, and you are not required to perform any exceptional effort.

SpatiaLite is absolutely *free* (in both meanings: it's *free as free beer*, and it's *free as free speech*): so you can get a quick glance at latest state-of-the-art Spatial SQL technologies in the easiest and simplest way.
And once you've got your own opinion on the basis of your direct first-hand experience, you can then decide by yourself if Spatial SQL can be useful (one way or the other) for your daily activities.

How this tutorial is structured: think of some ***Cookbook***. At first you are expected to simply acquire some limited and basic knowledge about:

- pottery and kitchen tools: *pans, bowls, pots, knives, spoons, whisks ....*
- commonly used ingredients: *eggs, fish, meat, vegetables, spices, seasonings ...*
- elementary cooking techniques: *boiling, grilling, roasting, deep-frying, creaming ...*

Once you have acquired such basic notions you are ready to confront yourself with simple but nutrient and tasty recipes: we can name this level as ***family cooking***
For many people this is their best effort: that's enough for them, and they'll quit at this point.

But many other will probably discover that after all cooking is funny and really interesting: so they'll surely ask for something much more sophisticated.
Any good cookbook will certainly end introducing several complex ***haute cuisine*** recipes.
May well be you'll never become a real ***chef de cuisine***, but you will amuse yourself anyway, and feel proud of your cooking skills.

SpatiaLite Cookbook
Table of contents

Kitchen tools and cooking techniques

Commonly used ingredients

Family cooking

Haute cuisine

Desserts, spirits, tea and coffee

Author: Alessandro Furieri a.furieri@lqt.it

This work is licensed under the Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license.

Permission is granted to copy, distribute and/or modify this document under the terms of the
GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Technical Introduction

2011 January 28

Quick Techical Intro: in technical terms ***SpatiaLite*** is a **Spatial DBMS** supporting international standards such as **SQL92** and **OGC-SFS**.

I suppose all the above acronyms sounds really obscure and (*maybe*) irritating to you.
Don't be afraid: very often obscure technical jargon hides really simple to understand concepts:

- a **DBMS** [*Database Management System*] is a software designed to store and retrieve arbitrary data in the most efficient and generalized way.
  Actually, lots and lots of huge and highly structured complex data.
- **SQL** [*Structured Query Language*] is a standardized language supporting DBMS handling:
  using *SQL statements* you can define how your data are organized.
  And you can insert, delete or update your data into the DBMS.
  Obviously, you can also retrieve (query) your data in a very flexible (and efficient) way.
- **OGC-SFS** [*Open Geospatial Consortium - Simple Feature Specification*] allows to extend the basic DBMS / SQL capabilities so to support a special **Geometry** data-type, thus allowing to deploy a so-called **Spatial DBMS**

SpatiaLite is widely based on the top of the very popular **SQLite**, a lightweight personal DBMS.
They act as a tight couple: *SQLite* implements a standard SQL92 data engine, while *SpatiaLite* implements the standard OGC-SFS core.
Using both them combined you'll then get a complete *Spatial DBMS*

SQLite/SpatiaLite aren't based on the most common **client-server** architecture: they adopt a simpler **personal** architecture. i.e. the whole SQL engine is directly embedded within the application itself.
This elementary simple and unsophisticated architecture widely simplifies any task related with database management: you can simply open (or create) a **database-file** exactly in the same way you are accustomed to follow when you open a text document or a spreadsheet. There is absolutely no extra-complexity implied in such tasks.
A complete **database** (*maybe, one containing several millions entities*) simply is an ordinary file. You can freely copy (or even delete) this file at your will without any problem.
And that's not all: such *database-file* supports a **universal** architecture, so you can transfer the whole *database-file* from one computer to a different one without any special precaution.
The origin and destination computer can actually use completely different operating systems: this has no effect at all, because database-files are **cross-platform portable**

Clearly, all this *simplicity* and *lightness* has a cost: SQLite/SpatiaLite support for concurrent multiple access is very rudimentary and poor.
This is what personal DB exactly means; the underlying paradigm is: **single user / single application / standalone workstation**

If supporting multiple concurrent access is a main goal for you, then SQLite/SpatiaLite aren't the better choice for your needing: a most complex *client-server* DBMS is then strongly required.
Anyway, SpatiaLite is very similar to **PostgreSQL/PostGIS** (a heavy-weighted client-server *open source* Spatial DBMS): so you can freely switch (in a relatively painless way) from the one to the other accordingly to your actual requirements, choosing each time the best tool to be used.

Some useful further references:

- SQLite home
- SpatiaLite home
- SQL as understood by SQLite (technical reference)
- SpatiaLite Manual (outdated version, but still userful)
- Old SpatiaLite Tutorial (outdated version, but still userful)

Getting started [installing the software]

2011 January 28

This one isn't a theory handbook: this one is a practical how-to tutorial for absolute beginners.
The underlying assumption is that you ignore everything about DBMS, SQL and even GIS.
So we'll begin downloading few resources from the WEB, and we'll then immediately start our operative session:

- first we'll download the **spatialite_gui** software: this one is a simple but powerful tool (supporting graphics, mouse and so on ..) allowing you to interact with a SpatiaLite database.
- then we'll download some publicly available dataset required to build the sample DB we'll use during our exercises.

Download the app

Start your web browser and go to SpatiaLite's home: http://www.gaia-gis.it/spatialite
The current version (January 2011) is v.2.4.0-RC4, and you can get executable binaries from: http://www.gaia-gis.it/spatialite-2.4.0-4/binaries.html

Quite obviously the download site layout changes from time to time, so the above URLs may become quickly outdated.
May be Linux users have to build binaries from themselves starting from sources: in such case read carefully the appropriate release notes before starting.

Download the sample dataset #1

The first dataset we'll use is the **Italian National Census 2001**, kindly released by **ISTAT** (*the Italian Census Bureau*).
Point your web browser at http://www.istat.it/ambiente/cartografia/ and then download the following files:

- Censimento 2001 - Regioni (*Regions*): http://www.istat.it/ambiente/cartografia/regioni2001.zip
- Censimento 2001 - Province (*Counties*): http://www.istat.it/ambiente/cartografia/province2001.zip
- Censimento 2001 - Comuni (*Local Councils*): http://www.istat.it/ambiente/cartografia/comuni2001.zip

Download the sample dataset #2

The second required dataset is **GeoNames**, a worldwide collection of Populated Places.
There are several flavors of this dataset: we'll use **cities-1000** (*any populated place into the word counting more than 1,000 peoples*).
http://download.geonames.org/export/dump/cities1000.zip

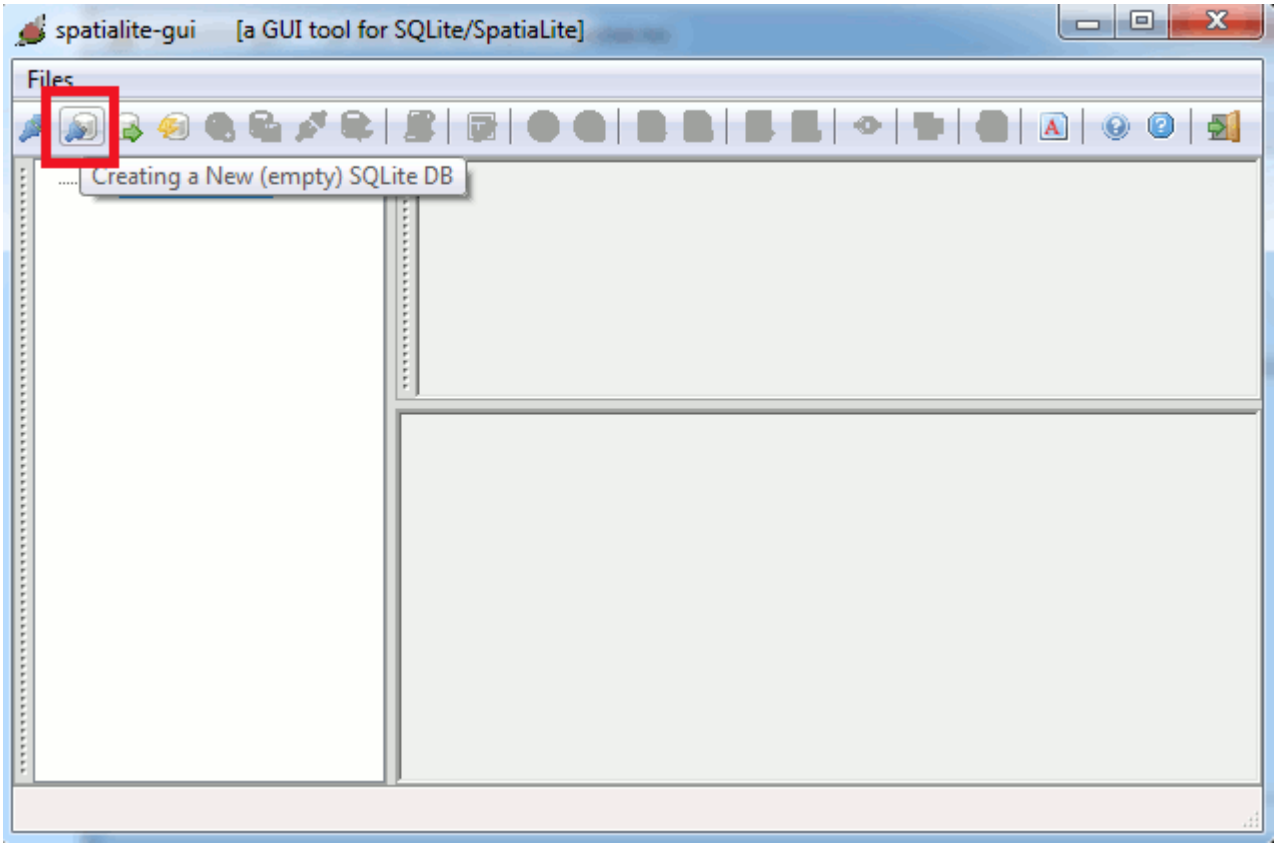
Launching the app

The **spatialite_gui** tool doesn't require any installation: simply *unzip* the compressed image you've just now downloaded, and click the launch icon. That's all.

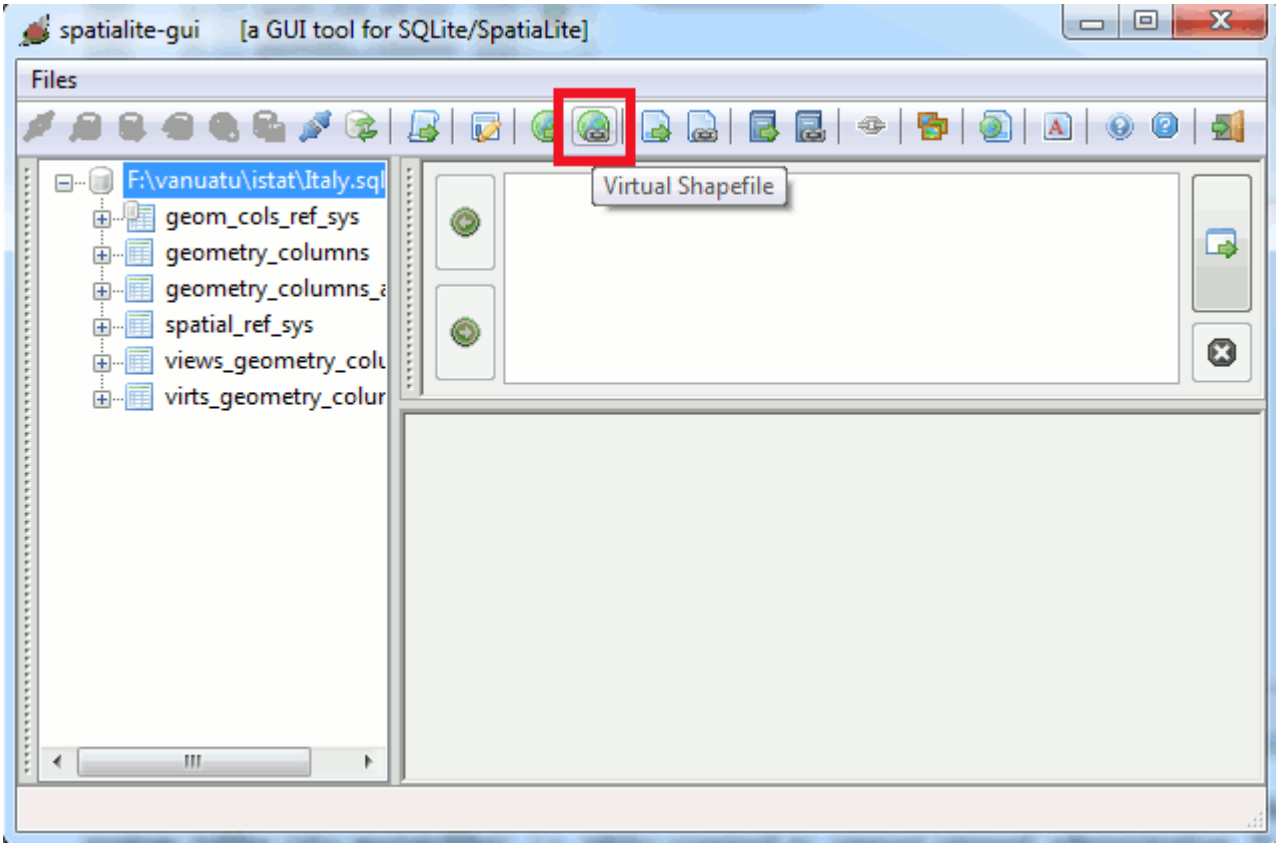
Building your first Spatial Database

2011 January 28

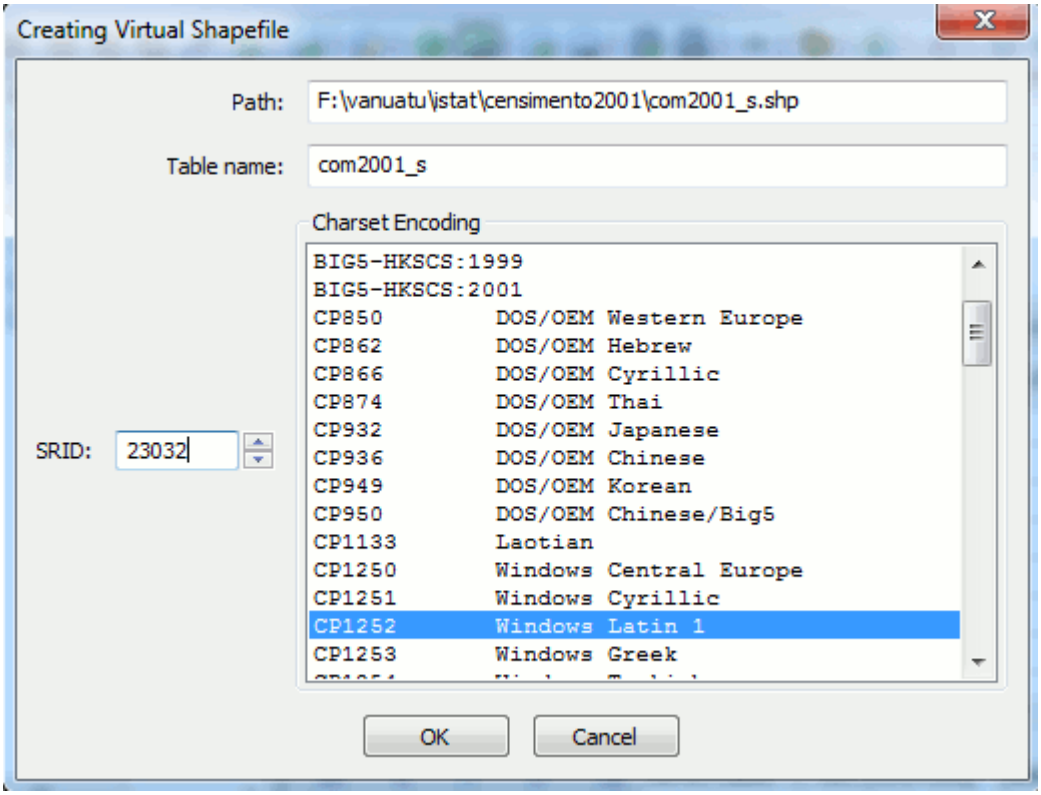
All right, you just started your first SpatiaLite working session: as you can easily notice, there is no DB currently connected.



so you'll now create / connect a new DB file: simply press the corresponding button from the tool bar (a platform standard *file open dialog* will soon appear) and set some file name.
Just for uniformity, please name this DB as **`Italy.sqlite`**

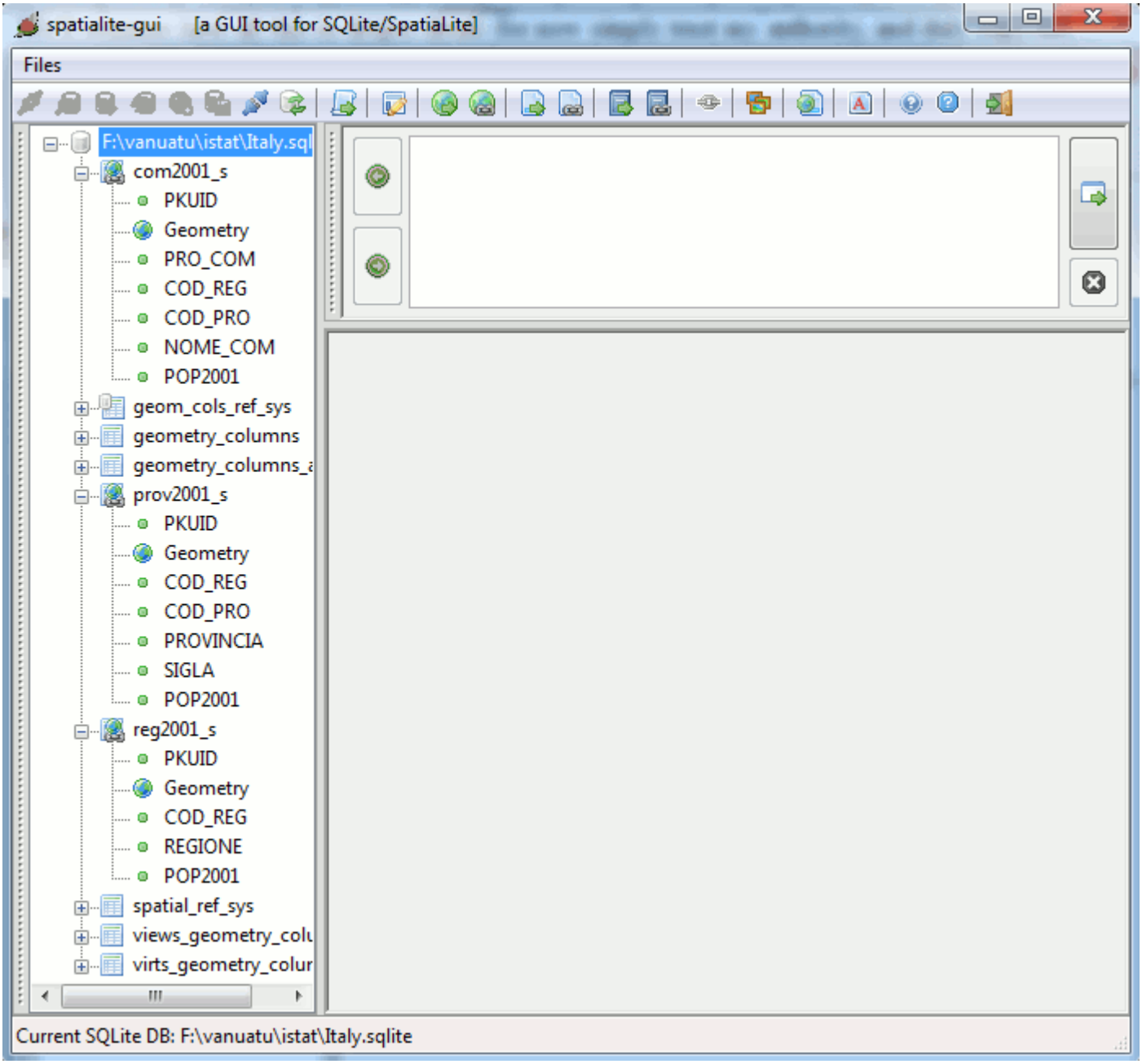


As you can notice, immediately after creation the DB already contains several tables: all them are **system tables** (aka **metatables**), i.e. tables required to support internal administration.
For now, simply ignoring them at all is the better choice to be done (*you are an absolute beginner, isn't ? be patient, please*).
Anyway, now you are connected to a valid DB, so you can now load the first dataset: press the **Virtual Shapefile** button on the toolbar, and then select the **`com2001_s`** file.

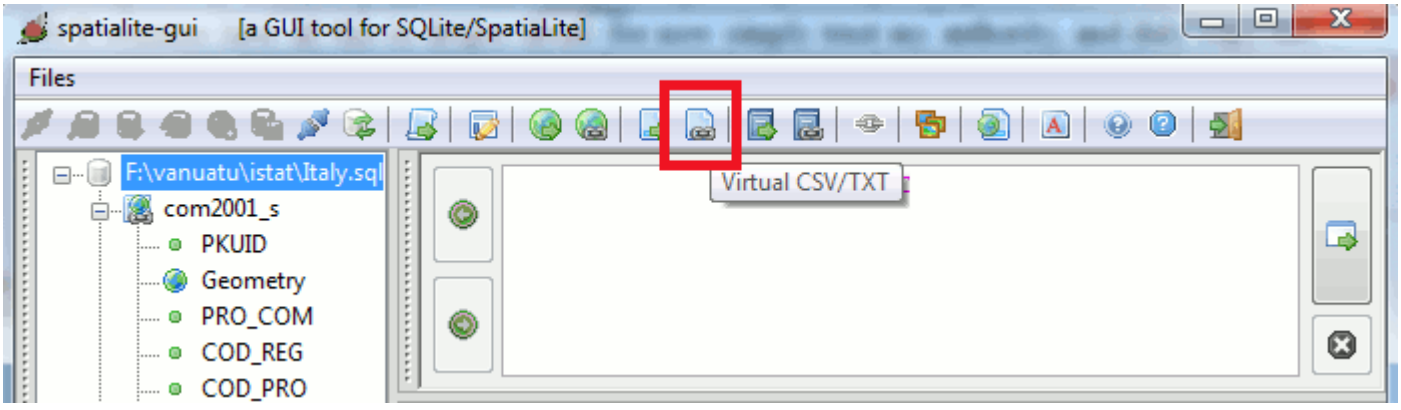


A dialog box will appear: please, select *exactly* the above shown settings and confirm.
We'll examine later all this in deeper detail: for now simply trust my authority, and duly copy the suggested values avoiding to understand at all: that's *black magic* for now, and that's all.

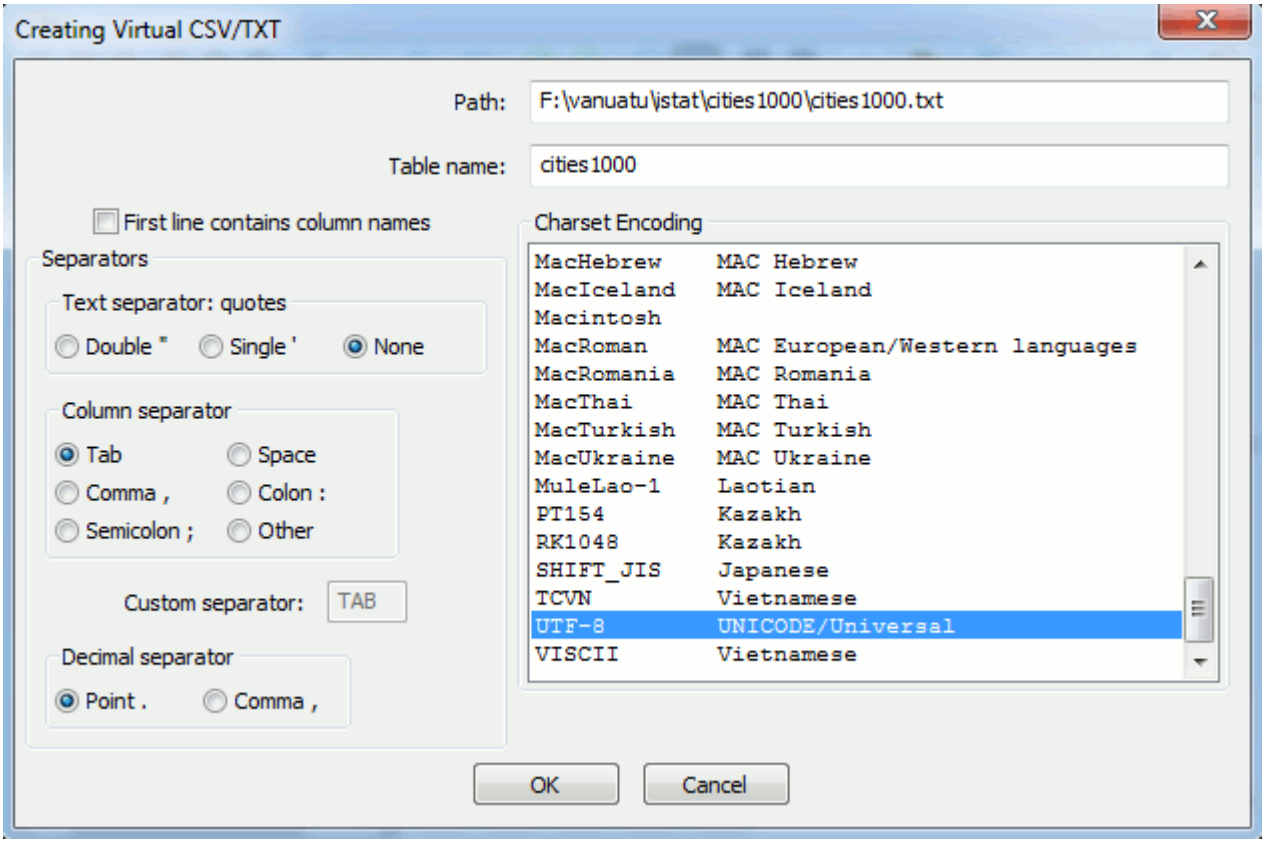
Once you've loaded the **`com2001_s`** dataset you can then continue (always using the same settings) and load both **`prov2001_s`** and **`reg2001_s`** files.



Your database will now look like this: using the left-sided tree-view control is really easy checking tables (and columns within each table).



You are now ready to complete the initial DB setup: press the **Virtual CSV/TXTe** button on the toolbar, and then select the **`cities1000.txt`** file.



A dialog box will appear: please, select *exactly* the above shown settings and confirm.
This dialog box strongly resembles the one you've already used in order to connect Virtual Shapefiles, but isn't identical. In this case too we'll examine later any related detail.
All right: now you have three datasets ready to be queried: but it's now time to explain better what we where doing in the above steps.

About ESRI Shapefiles and Virtual Tables

2011 January 28

What's a Shapefile ?

Shapefile is a plain, unsophisticated GIS (geographic data) file format invented many years ago by ESRI: although initially born in a proprietary environment, this file format has been later publicly disclosed and fully documented, so it's now really like an *open standard format*.

It's rather obsolescent nowadays, but it's universally supported. So it represents the *lingua franca* every GIS application can surely understand: and not at all surprisingly, SHP is widely used for cross platform neutral data exchange.

The name itself is rather misleading: after all, *Shapefile* isn't a simple file. At least three distinct files are required (identified by **.shp .shx .dbf** suffixes): if a single file is missing (*misnamed / misplaced / malformed / whatsoever else*), then the whole dataset is corrupted and completely not usable.

Some useful further references:

- http://en.wikipedia.org/wiki/Shapefile (simple introduction)
- http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf (technical reference)

What's a Virtual Shapefile (and Virtual Tables) ?

SpatiaLite supports a Virtual Shapefile driver: i.e. it has the capability to support SQL access (*read-only* mode) for an external *Shapefile*, with no need to load any data within the DB itself.
This is really useful during any preliminary database construction step (as in our case).
SQLite/SpatiaLite supports several other different Virtual drivers, such as the Virtual CSV/TXT, the Virtual DBF and so on ...

Anyway, **be warned**: Virtual Tables suffer from several limitations (*and are often much slower than using internal DB storage*), so **they are not at all intended for any serious production task**.

About Charset encodings

2011 January 28

What's a Charset encoding ? (*and why the hell have I to take care of such nasty things ?*)

Very simply said: any computer really is a stupid machine based on a messy bunch of crappy silicon: it has some intrinsic capability to understand simple arithmetic and boolean algebra, but it's absolutely unable to understand text.
You can easily had stored somewhere in your brain the misleading notion that a computer can actually handle text, but that's not exactly the truth.
To be most precise, it's rather a *hocus-pocus* finalized to mock you and your limited senses, dumb human being: any computer simply handles digits, but peripheral devices (*screen, keyboard, printer* ..) are purposely designed to give you the (*illusory*) impression that your PC actually understands text.

All this is an absolutely ***conventional process***: you and your PC must agree about some ***correspondence table*** to be used in order to translate obscure digit sequences into readable words. In technical terms, such a *conventional correspondence table* is known as a **Charset Encoding**.
How many different alphabets are used into the Earth ? lots and lots ... *Latin, Greek, Cyrillic, Hebrew, Arabic, Chinese, Japanese* and many others ... and accordingly to this, lots and lots of different Charset Encodings has been defined during the years (*you know: electronic / computer industry is fond of conflicting standard uncontrolled proliferation*).

SQLite/SpatiaLite internally always uses the **UTF-8** encoding, which is **universal** (i.e. you can safely store any known alphabet within the same DB at the same time): unhappily, *Shapefiles* (and many other datasets, such as SVC/TXT files) aren't UTF-8 based (they use some national encoding instead), so you are forced to explicitly select the Charset encoding to be used each time you have to import (or export) any data. I'm really sorry for this, but that's reality.

Anyway, consider all this not as a complication, but as a big resource: this way you'll be correctly able to import/export any arbitrary dataset coming from exotic not-latin countries such as Israel, Japan, Vietnam, Greece or Russia.
And after all, being able to display multi-alphabet text strings (such as the following one) can make your friends to become green with envy: **Roma,??µ?,???,??**

Some useful further references:

- http://en.wikipedia.org/wiki/Character_encoding (simple introduction)
- http://www.gnu.org/software/libiconv/ (technical reference)

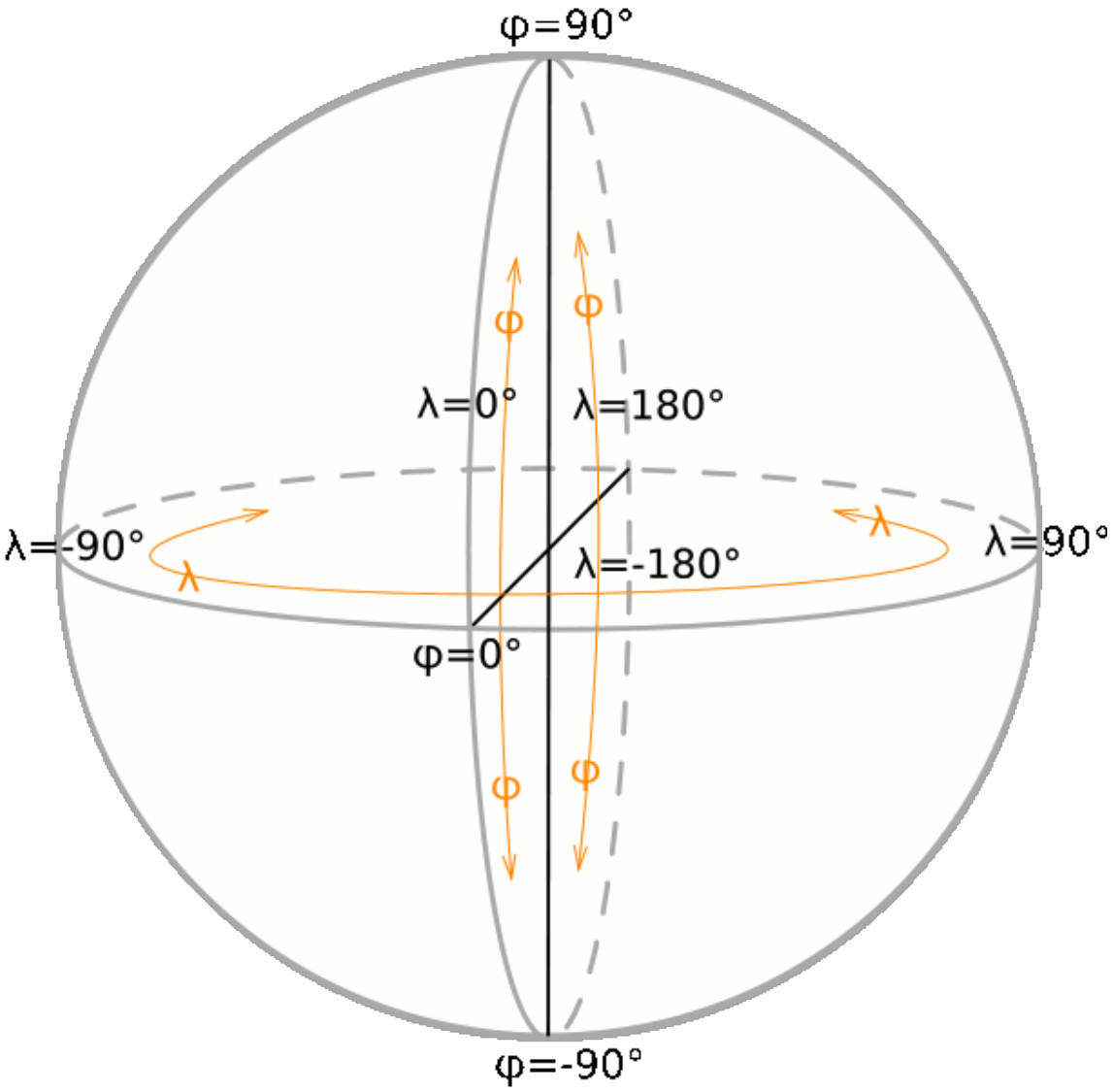
what's this SRID stuff ? ... I've never heard this term before now ...

2011 January 28

What's this SRID stuff ? ... *I've never heard this term before now* ...

Planet Earth is a ***sphere*** ... not exactly, planet Earth has an ***ellipsoidal*** shape (*slightly flattened at poles*) ...
oh no, that's absolutely wrong: planet Earth hasn't a geometric regular shape, it actually is a ***geoid***

All the above assertions can be assumed to be true, but at different approximation levels.
Near the Equator differences between a sphere and an ellipsoid are rather slight and quite unnoticeable; but neat both Poles such differences becomes greater and most easily appreciable.
For many practical purposes differences between an ellipsoid and a geoid are very slim: but for long range aircraft navigation (or even worse, for satellite positioning), this is too much simplistic and unacceptably approximate.

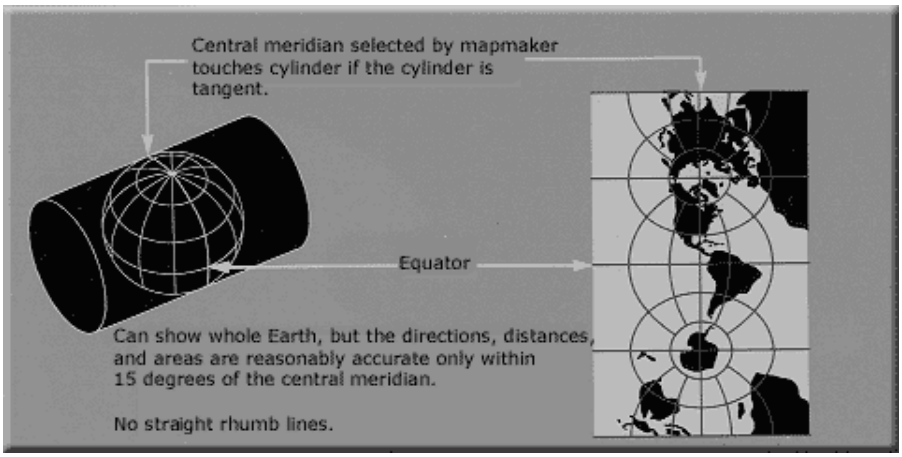


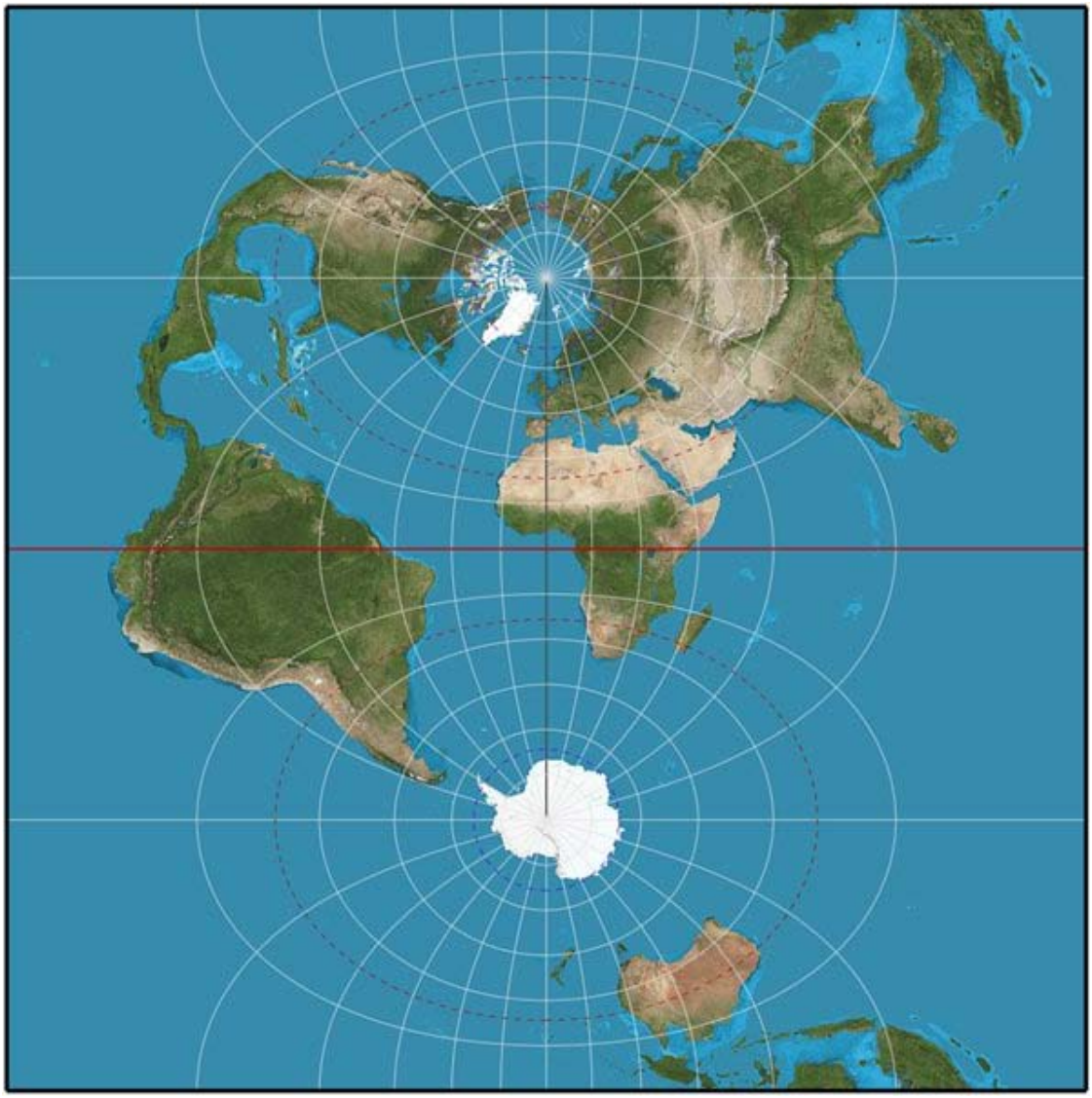
Anyway, whatsoever could be the real shape of the Earth, position of each point on the planet surface can precisely determined simply measuring two **angles**: **longitude** and **latitude**.
In order to set a complete **Spatial Reference System** [aka ***SRS***] we can use the ***Poles*** and the ***Equator*** (*which after all are outstanding places by intrinsic astronomic properties*): choosing a ***Prime Meridian*** on the other side is absolutely conventional: but since many centuries (*Britannia rule the waves* ...) adopting the ***Greenwich Meridian*** is an obvious choice.

Any SRS based on **long-lat** coordinates is known as a **Geographic System**. Using a Geographic SRS surely grants you maximum precision and accuracy: but unhappily this fatally implies several undesirable side-effects:

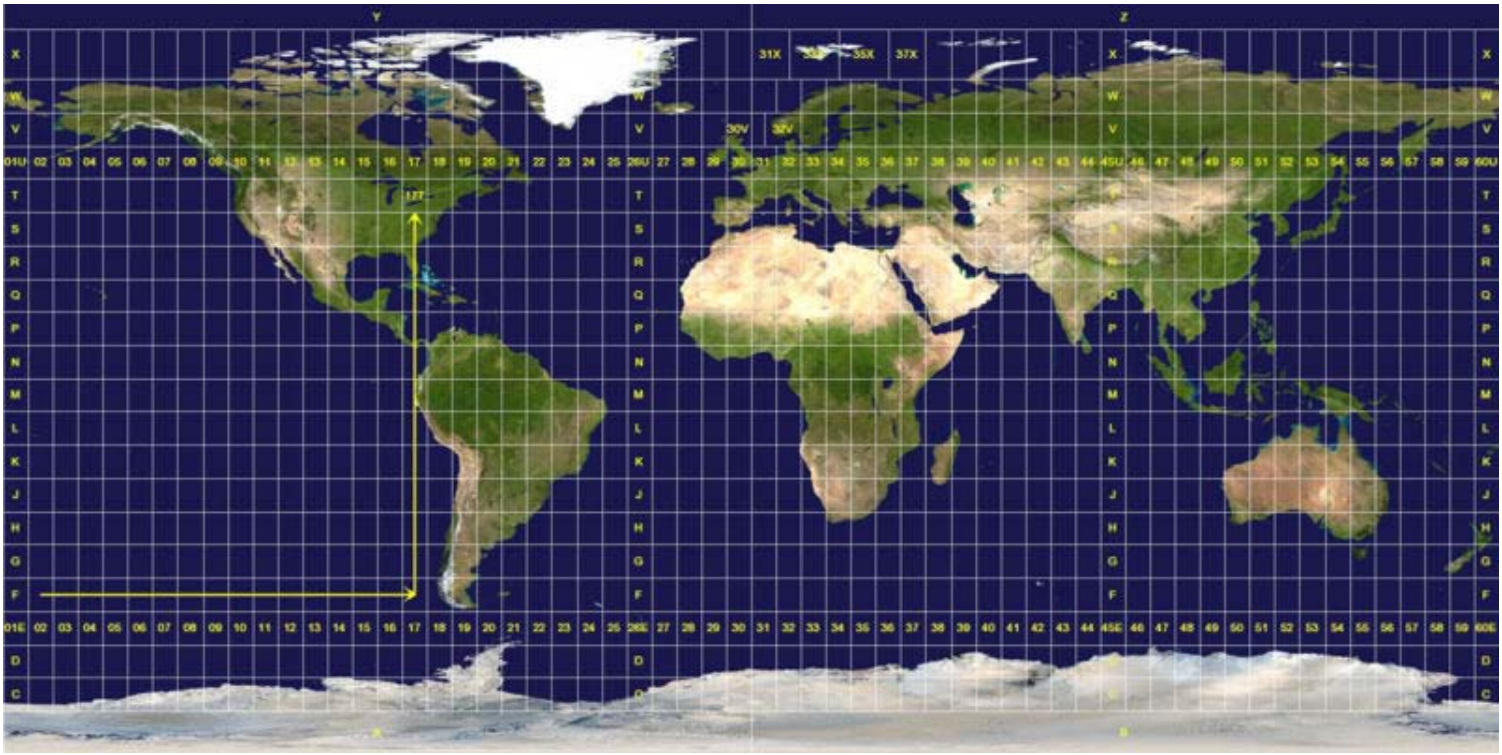
- paper sheets (and monitor screens) are absolutely flat; they don't look at all like a sphere
- using ***angles*** makes measuring distances and areas really difficult and counter-intuitive.

So since many centuries cartographers invented several (*conventional*) systems enabling to represent spherical surfaces into a flatten plane: none of them all is *the best one*.
All them introduce some degree of approximation and deformation: choosing the one or the other implies an absolutely *arbitrary* and *conventional* process: a map projection good to represent small Earth's portions can easily be awful when used to represent very wide territories, and *vice versa*.
We'll quickly examine the **UTM** [*Universal Transverse Mercator*] map projection, simply because it's really often used.





Apparently this map projection introduces severe and not acceptable deformations: but when you carefully focus your attention on the narrow central fuse, you'll immediately recognize that UTM allows to get a nearly perfect planar projection of excellent quality.
Anyway all this has a price: the central fuse has to be really narrow (let say, it will span only few degrees on both sides). As the fuse becomes wider, as much more deformations will become stronger and more evident.



Accordingly to all the above considerations, UTM defines 60 standard zones, each one covering exactly 6 longitude degrees.
Merging together two adjacent fuses (12 degrees) obviously reduces accuracy, but is still acceptable for many practical purposes: exceeding this limit produces really low-quality results, and has to be absolutely avoided.

Attempting to standardize the chaos

During the past two centuries every National State has introduced at least one (and very often, more than one) map projection system and related SRS: the overall result is absolutely chaotic (and really painful to be handled).

Happily, an international standard is widely adopted so to make easier correctly handling map SRS: the **European Petroleum Survey Group** [***EPSG***] maintains a huge worldwide dataset of more than 3,700 different entries.
Many of them are nowadays obsolete, and simply play a historical role; many others are only useful in very limited national boundaries.
Anyway, this one is an absolutely impressive collection.
And each single entry within the EPSG dataset is uniquely identified by its **numeric ID** and **descriptive name**, so to avoid any possible confusion and ambiguity.

Any Spatial DBMS requires some **SRID-value** to be specified for each Geometry: but such SRID simply is a ***Spatial Reference ID***, and (*hopefully*) coincides with the corresponding **EPSG ID**

Just in order to help you understand better this SRID chaos, this is a quite complete list of SRIDs often used in a (*small*) Nation such as **Italy**:

EPSG SRID	Name	Notes
4326	WGS 84	Geographic [*long-lat*]; worldwide; used by GPS devices
3003 3004	Monte Mario / Italy zone 1 Monte Mario / Italy zone 2	obsolete (1940) but still commonly used
23032 23033	ED50 / UTM zone 32N ED50 / UTM zone 33N	superseded and rarely used: European Datum 1950
32632 32633	WGS 84 / UTM zone 32N WGS 84 / UTM zone 33N	WGS84, adopting the planar UTM projection
25832	ETRS89 / UTM zone 32N	

25833	ETRS89 / UTM zone 33N	enhanced evolution of WGS84: official EU standard

And the following examples may help to understand even better:

Town	SRID	Coordinates	
		X (longitude)	Y (latitude)
Roma	4326	12.483900	41.894740
	3003	1789036.071860	4644043.280244
	23032	789036.071860	4644043.280244
	32632	789022.867800	4643960.982152
	35832	789022.867802	4643960.982036
Milano	4326	9.189510	45.464270
	3003	1514815.861095	5034638.873050
	23032	514815.861095	5034638.873050
	32632	514815.171223	5034544.482565
	35832	514815.171223	5034544.482445

As you can easily notice:

- WGS84 [4326] coordinates are expressed in *decimal degrees*, because this one is a ***Geographic System*** directly based on long-lat angles.
- on the other side any other system adopts coordinates expressed in *meters*: all them are ***projected*** aka ***planar*** systems.
- Y-values look very similar for every planar SRS: that's not surprising, because this value simply represents the distance from the Equator.
- X-values are more dispersed, because different SRSes adopt different ***false easting*** origins: i.e. they place their Prime Meridian in different (*conventional*) places.
- Anyway, any UTM-based SRS gives very closely related values, simply because all them share the same ***UTM zone 32*** definition.
- The (*small*) differences you can notice about different UTM-based SRSes can be easily explained: UTM zone 32 is always the same, but the underlying ***ellipsoid*** changes each time.
  Getting a precise measure for ***ellipsoid's axes*** isn't an easy task: and obviously during the time several increasingly better and most accurate estimates has been progressively adopted.

Distance intercurring between Roma and Milano	
SRID	**Calculated Distance**
4326	4.857422
3003	477243.796305
23032	477243.796305
32632	477226.708868
35832	477226.708866
Great Circle	477109.583358
Geodesic	477245.299993

And now we can examine how using different SRSes affects distances:

- Using WGS84 [4326] *geographic, long-lat* coordinates we'll actually get a measure corresponding to an ***angle*** expressed in **decimal degrees**. [*not so useful, really ...*]
- any other SRS will return a distance measure expressed in **meters**: anyway, as you can easily notice, figures aren't exactly the same.
- **Great Circle** distances are calculated assuming that the Earth is exactly a sphere: and this one obviously is the worst estimate we can get.
- on the other side **Geodesic** distances are directly calculated on the reference Ellipsoid.

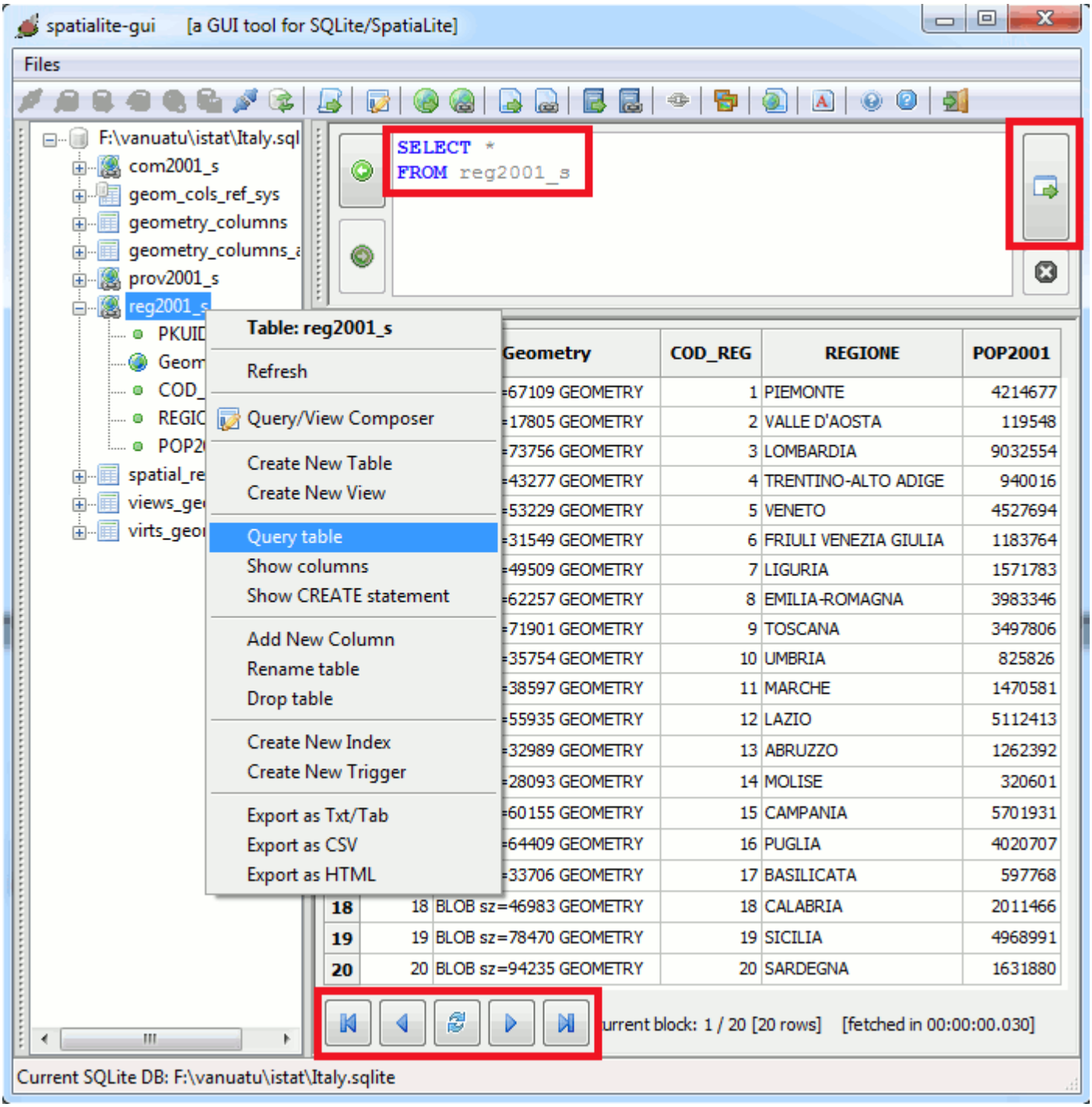
Conclusion:: **Thou shall not have exact measures**

But this isn't at all surprising in physical and natural sciences: any measured value is intrinsically affected by errors and approximations.
And any calculated value will be inexorably affected by rounding and truncation artifacts.

So absolutely exact figures simply doesn't exist in the real world: you have to be conscious that you can simply get some more or less approximated value.
But at least, you can take care to properly reduce such approximations in the best possible way.

Executing your first SQL queries

2011 January 28

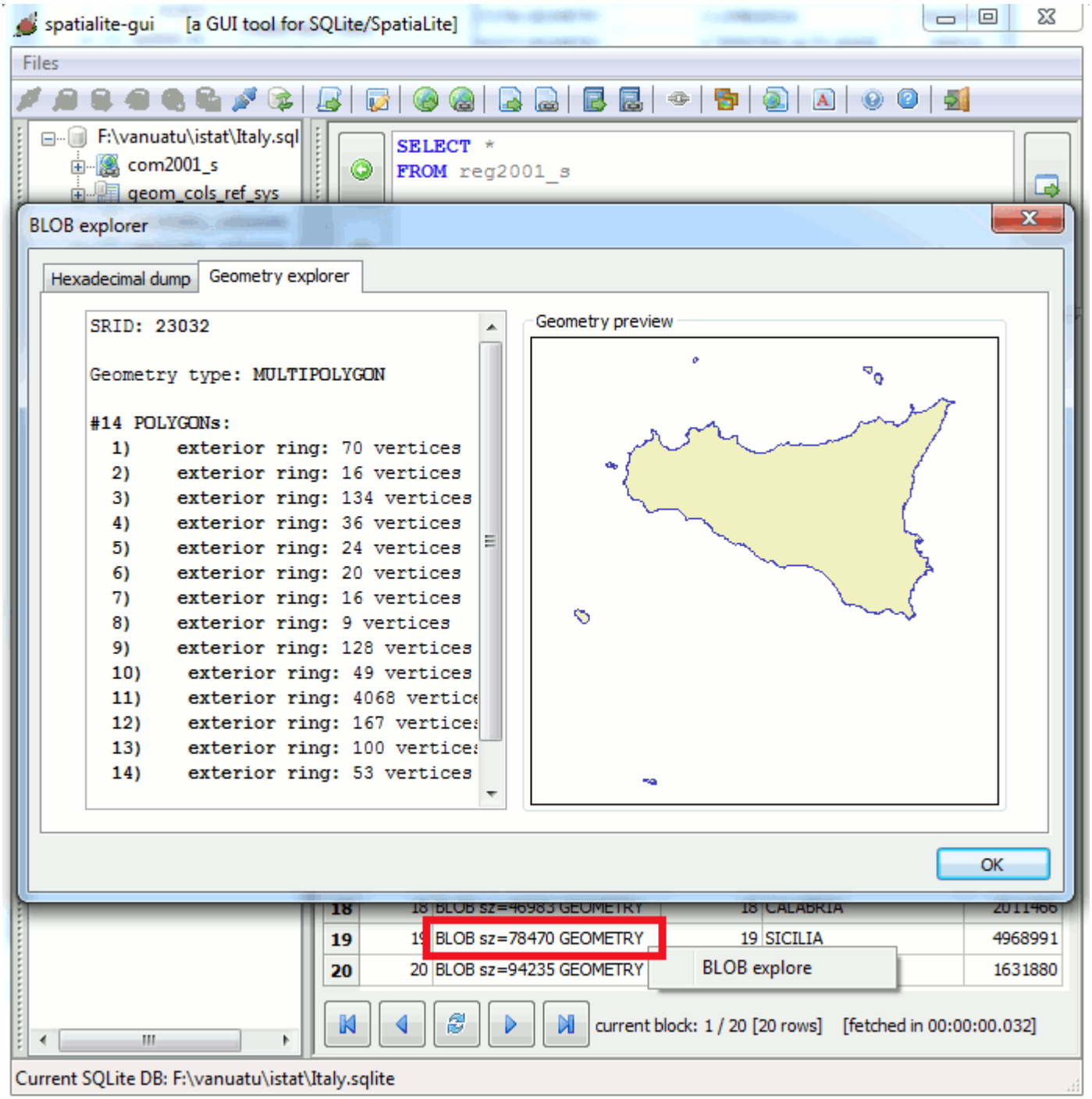


You can follow two different approaches in order to query a DB table:

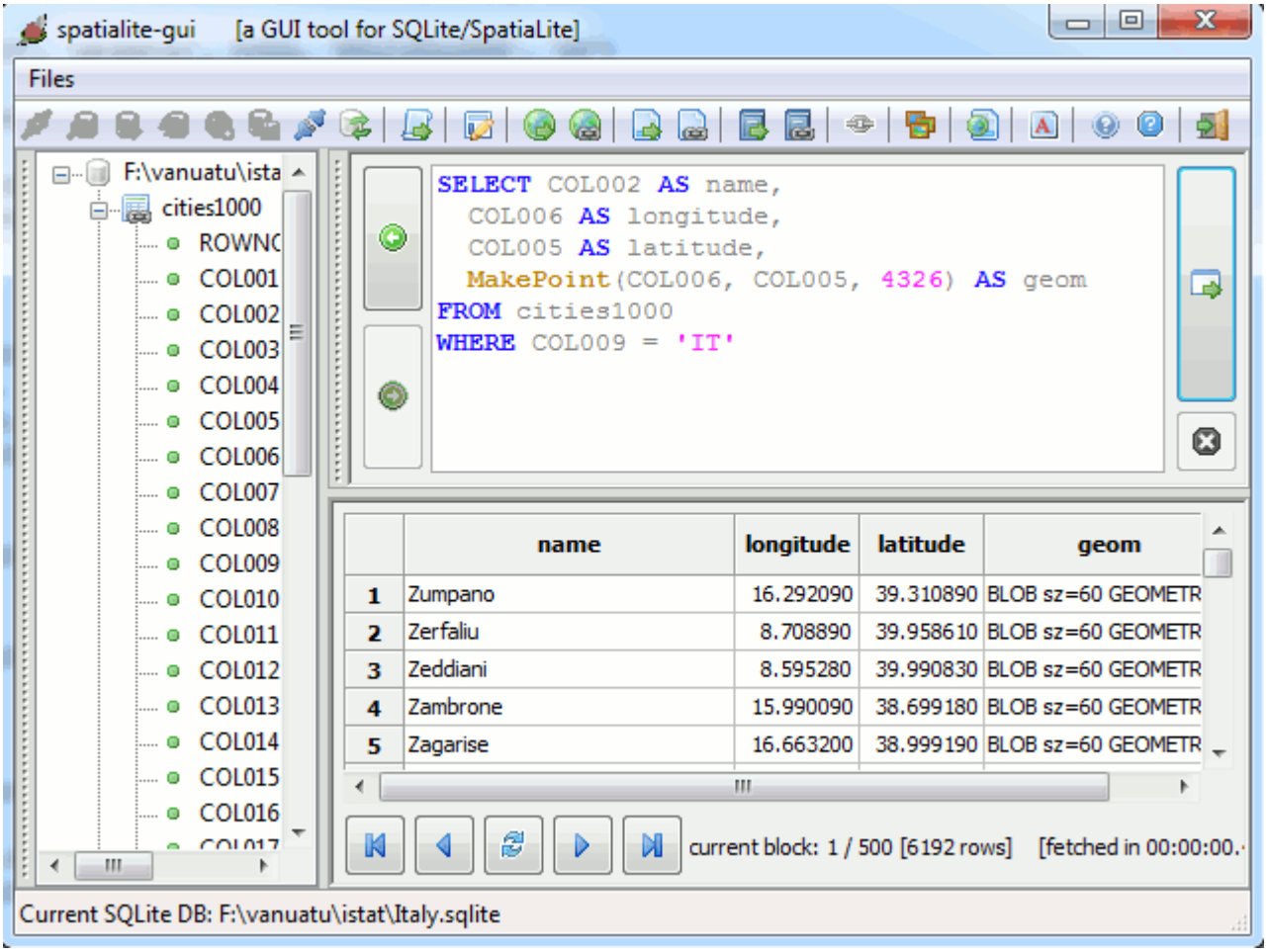
1. you can lazily use the **Query Table** menu item.
   a. this surely is quickest and easiest way, completely user friendly.
   b. you simply have to click the mouse's right button over the required table, so to make the context menu to be shown.

c. then you can simply use the lowermost buttons in order to scroll the result-set back and forth at your will.
d. anyway, this approach is rather mechanical, and doesn't allow you to exploit SQL at the best of its powerful capabilities.

2. alternatively you can hand-write any arbitrary *SQL statement* into the uppermost pane, then pressing the **Execute** button.
   a. this one is the hardest way: you are responsible for what you are doing (*or even misdoing* ...)
   b. but this way you can take full profit of the impressive SQL unconstrained fire power.



You surely noticed in the above illustration that Geometry columns simply reports an anonymous *BLOB GEOMETRY*: this is far to be satisfactory.
But you can get a much richer preview of any Geometry simply clicking the mouse's right button over the corresponding value, then selecting the **BLOB explore** menu item.




```
SELECT COL002 AS name,
  COL006 AS longitude,
  COL005 AS latitude,
  MakePoint (COL006, COL005, 4326) AS geom
FROM cities1000
WHERE COL009 = 'IT';
```

You can test how *free-hand* SQL works using this SQL statement: simply ***copy*** the above SQL statement, then ***paste*** into the query pane, and finally press the ***Execute*** button.
Just a quick and fast explanation:

- the **`cities1000`** table contains any populated place in the world
- there are lots of columns in this table, and their names are rather obscure
  (*you can find a full documentation for this at* http://www.geonames.org/)
  but you can simply trust to my authority just for now.
- **`COL002`** contains the **name** of each populated places.
- **`COL006`** contains the corresponding **longitude** (*expressed in decimal degrees*).
- **`COL005`** is the corresponding **latitude**
- **`MakePoint()`** is a Spatial function building a point-like Geometry from corresponding coordinates.
- **`COL009`** contains the **Country code**
- very simply said, the **`WHERE`** clause will filter the result-set so to exclude all places outside Italy.

All right, you are now supposed to be able to really start working seriously.
In the next slides we'll start exploring the mystery world of SQL and Spatial SQL.

Basics about SQL queries

2011 January 28

The following SQL queries are so elementary simple that you can directly verify the results by yourself.
Simply follow any example executing the corresponding SQL statement (using **copy&paste**).

```
SELECT *
FROM reg2001_s;
```

This one really is the *pons asinorum* of SQL: all columns for each row of the selected table will be dumped following a random order.

```
SELECT pop2001, regione
FROM reg2001_s;
```

You aren't necessarily obliged to retrieve every column: you can explicitly choose which columns have to be included into the result-set, establishing their relative order.

```
SELECT Cod_rEg AS code, REGIONE AS name,
  pop2001 AS "population (2001)"
FROM reg2001_s;
```

You can set a most appropriate and intelligible name for each column, if you feel this is appropriate.
this example shows two important aspects to be absolutely noticed:

- SQL names are *case-insensitive*: **`REGIONE`** and **`regione`** refers the same column.
- SQL names cannot contain ***forbidden characters*** (such as *spaces*, *brackets*, *colons*, *hyphens* and so on).
  And they cannot correspond to any ***reserved keyword*** (i.e. you cannot define a column named e.g. **`SELECT`** or **`FROM`**, because they shadow SQL commands)
- anyway you can explicitly ***mask*** any *forbidden* name, so to make it become fully legal.
  In order to apply such masking, you simply have to enclose the full name within **double quotes**.
- you can obviously indiscriminately double quote every SQL name: this is completely harmless, but not strictly required.
- in the (*extremely rare*, *but not impossible*) case your *forbidden* name already contains one or more double quote characters, you must insert an extra double quote for each one of them.
  e.g.: **`A"Bc"D`** has to be correctly masked as: **`"A""Bc""D"`**

```
SELECT COD_REG, REGIONE, POP2001
FROM reg2001_s
ORDER BY regione;
```

SQL allows you to order rows into the result-set in the most convenient way for your purposes.

```
SELECT COD_REG, REGIONE, POP2001
FROM reg2001_s
ORDER BY POP2001 DESC;
```

You can order in **`ASC`**ending or **`DESC`**ending order at your will: the **`ASC`** qualifier is usually omitted, simply because this one is the default ordering.

- Ascending order means A-Z for text values: and from lesser to greater for numeric values.

Descending order means Z-A for text values: and from greater to lesser for numeric values.

```
SELECT COD_PRO, PROVINCIA, SIGLA
FROM prov2001_s
WHERE COD_REG = 9;
```

Using the **WHERE** clause you can restrict the range: only rows satisfying the **WHERE** clause will be placed into the result-set: in this case, the list of Counties belonging to Tuscany Region will be extracted.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE COD_PRO = 48;
```

Same as above: this time the list of Local Councils belonging to the Florence County will be extracted.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE COD_REG = 9 AND POP2001 > 50000
ORDER BY POP2001 DESC;
```

You can combine more conditions under the same **WHERE** clause: this time the list of the most populated Local Councils (> 50,000 peoples) belonging to Tuscany Region will be extracted.
And the result-set will be ordered accordingly to decreasing population.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com = 'ROMA';
```

You can obviously use text strings as comparison values: in *pure* SQL any text string has to be enclosed within **single quotes**.
[*SQLite is smart enough to recognize **double quoted** text strings as well, but I strongly discourage you to adopt this bad style as your preferred one*].
Please note well: string values comparisons for SQLite are always ***case-sensitive***.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com = 'L''AQUILA';
```

When some text string contains an *apostrophe*, you have to apply *masking*.
An extra **single quote** is required to mask every apostrophe withing the text string: e.g.: `REGGIO NELL'EMILIA` has to be correctly masked as: `'REGGIO NELL''EMILIA'`

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com LIKE 'roma';
```

You can use the *approximate evaluation* operator **LIKE** to make text comparisons to become ***case-insensitive***.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com LIKE '%maria%';
```

And you can use the operator **LIKE** so to apply partial text comparison, using **%** as a wild-card: this query will extract any Local Council containing the sub-string `'maria'` within its name.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE nome_com IN ('ROMA', 'MILANO', 'NAPOLI');
```

Sometimes may be useful using a list of values, as the one shown above.

```
SELECT PRO_COM, NOME_COM, POP2001
FROM com2001_s
WHERE POP2001 BETWEEN 1990 AND 2010;
```

Another not often used, but sometimes useful comparison criterion is the one to set a range of values to be checked.

```
SELECT PROVINCIA, SIGLA, POP2001
FROM prov2001_s
WHERE COD_REG IN (9, 10, 11, 12)
  AND SIGLA NOT IN ('LI', 'PI')
  AND (POP2001 BETWEEN 300000 AND 500000
    OR POP2001 > 750000);
```

Using SQL you can set any kind of complex **WHERE** clause: there are no imposed limits.
And this one is a really fantastic feature, disclosing potentially infinite scenarios.
Just a short explanation: the previous query will:

- include any County in Central Italy (Regions: Tuscany, Umbria, Marche and Lazio)
- excluding the Livorno and Pisa Counties
- then filtering Counties by population:
  - including the ones within the range 300,000 to 500,000
  - including as well the ones exceeding 750,000 peoples

```
SELECT PROVINCIA, SIGLA, POP2001
FROM prov2001_s
WHERE COD_REG IN (9, 10, 11, 12)
  AND SIGLA NOT IN ('LI', 'PI')
  AND POP2001 BETWEEN 300000 AND 500000
  OR POP2001 > 750000;
```

Please note well: in SQL the logical connector **OR** has a very low priority.
Check by yourself: omitting to properly enclose the OR-clause within brackets produces very different results, isn't ?

```
SELECT *
FROM com2001_s
LIMIT 10;
```

There is a last sometimes useful **SELECT** clause to explain: using **LIMIT** you can set the maximum number of rows to be extracted into the result-set
(*very often you aren't actually interested into reading a very densely populated table as a whole: a shorted preview will easily be enough in many cases*).

```
SELECT *
FROM com2001_s
LIMIT 10 OFFSET 1000;
```

And that's not all: SQL doesn't constraints you to read a limited row-set necessarily starting from the beginning: you can place the start-point at your will, simply using **OFFSET** combined with **LIMIT.**

Learning **SQL** isn't so difficult after all.
There are very few **keywords**, language syntax is notably regular and predictable, and query statements are designed to resemble *plain English* (as much as possible ...).
Now you are supposed to be able to attempt writing (*simple*) SQL query statements by yourself.

Understanding aggregate functions

2011 January 28

We've seen since now how SQL allows to retrieve single values on a row-per-row basis.
A different approach is supported as well, one allowing to compute ***total values*** for the whole table, or for group(s) of selected rows.
This implies using some special functions, known as **aggregate functions**.

```
SELECT Min(POP2001), Max(POP2001),
  Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s;
```

This query will return a single row, representing something like a summary for the whole table:

- the **`Min()`** function will return the minimum value found into the given column
- the **`Max()`** function will return the maximum value found into the given column
- the **`Avg()`** function will return the average value for the given column
- the **`Sum()`** function will return the total corresponding to the given column
- the **`Count()`** function will return the number of rows (entities) found

```
SELECT COD_PRO, Min(POP2001), Max(POP2001),
  Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s
GROUP BY COD_PRO;
```

You can use the **`GROUP BY`** clause in order to establish a more finely grained aggregation by sub-groups.
This query will return distinct totals for each County.

```
SELECT COD_REG, Min(POP2001), Max(POP2001),
  Avg(POP2001), Sum(POP2001), Count(*)
FROM com2001_s
GROUP BY COD_REG;
```

And you can obviously get totals for each Region simply changing the **`GROUP BY`** criterion.

```
SELECT DISTINCT COD_REG, COD_PRO
FROM com2001_s
ORDER BY COD_REG, COD_PRO;
```

There is another different way to aggregate rows: i.e. using the **`DISTINCT`** clause.
Please note well: this isn't absolutely the same as using a **`GROUP BY`** clause:

- **`DISTINCT`** simply suppresses any duplicate row, but has nothing to do with aggregate functions.
- **`GROUP BY`** is absolutely required each time you have to properly control aggregation.

Your first Spatial SQL queries

2011 January 28

SpatiaLite is a **Spatial DBMS**, so it's now time to perform some **Spatial SQL** query.
There isn't absolutely nothing odd in Spatial SQL: it basically is exactly as standard SQL, but it supports the exotic data-type Geometry.
Usually you cannot directly query a Geometry value (as we've already seen they simply are a meaningless BLOB): you are expected to use some appropriate **spatial function** to access a Geometry value in a meaningful way.

```
SELECT COD_REG, REGIONE, ST_Area(Geometry)
FROM reg2001_s;
```

The **`ST_Area()`** function is one of such Spatial functions; usually you can easily recognize any Spatial function, simply because all them are **ST_** prefixed.
This one is not an absolute rule, anyway: SpatiaLite is able to understand the *alias* name **`Area()`** to identify the same function.
As the name itself states, this function computes the surface of the corresponding Geometry.

```
SELECT COD_REG AS code,
  REGIONE AS name,
  ST_Area(Geometry) / 1000000.0 AS "Surface (sq.Km)"
FROM reg2001_s
ORDER BY 3 DESC;
```

As you surely noticed, the first query returned very high figures: this is because the current dataset uses **meters** as length unit, and consequently surfaces are measured in **m²**.
But we simply have to apply an appropriate scale factor to get the most usual **km²** units.
Please note two SQL features we are introducing for the first time:

- SQL isn't constrained to directly return the column value into the result-set: you can freely define any valid arithmetic expression as required.
- Referencing a complex expression into some **`ORDER BY`** clause isn't too much practical:
  but you can easily identify any column using its ***relative position*** (first column has index 1, and so on).

```
SELECT COD_REG AS code,
  REGIONE AS name,
  ST_Area(Geometry) / 1000000.0 AS "Surface (sq.Km)",
  POP2001 / (ST_Area(Geometry) / 1000000.0)
    AS "Density: Peoples / sq.Km"
FROM reg2001_s
ORDER BY 4 DESC;
```

And you can perform even more complex calculations in SQL.
This query will compute the **population density** (measured as **peoples / km²**).

All right, you have now acquired a basic SQL / Spatial SQL knowledge.
You are now ready to confront yourself with most complex and powerful queries: but this requires building a serious database.
Do you remember ? for now we where simply using Virtual Shapefiles tables; i.e. the faint imitation of real Spatial tables (*internally stored*).
So during the next steps we'll first create and populate a **well designed DB** (not a so trivial task), and then we'll come again to see most complex and sophisticated SQL queries.

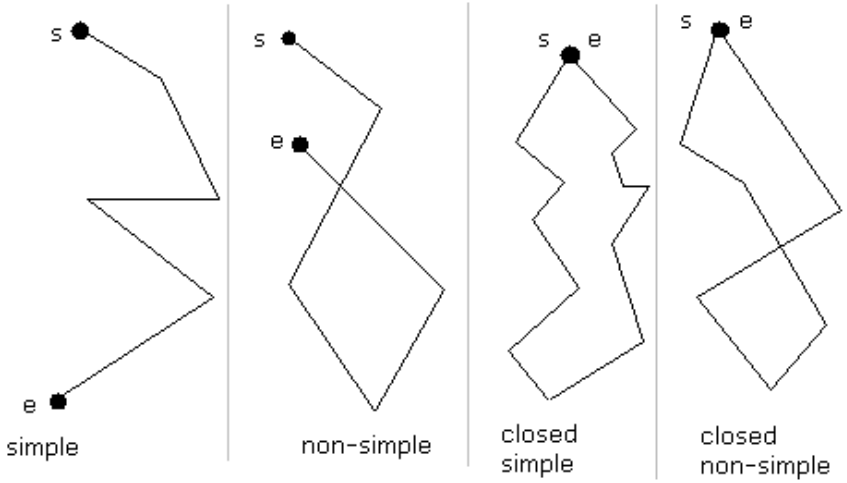
More about Spatial SQL: WKT and WKB

2011 January 28

SpatiaLite supports a **`Geometry`** data type conformant to the international standard **OGC-SFS** (*Open Geospatial Consortium - Simple Feature SQL*).
http://www.opengeospatial.org/standards/sfs

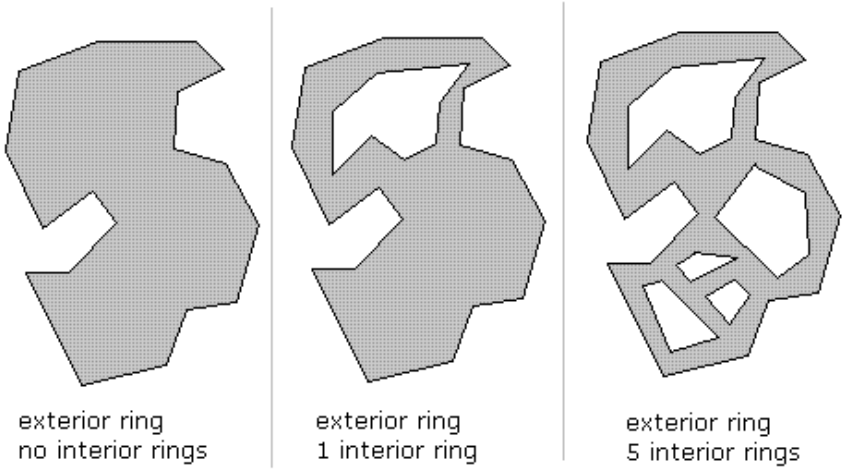
`Geometry` is an ***abstract*** data type with seven related ***concrete*** *sub-classes*.
You cannot directly instantiate a **`Geometry`** (*because this one is and* ***abstract*** *class, and doesn't corresponds to any actual implementation*): but you can freely instantiate any related **sub-class**.

Sub-Class	Example
`POINT`	● [x,y] POINT
`LINESTRING`	



LINESTRINGs

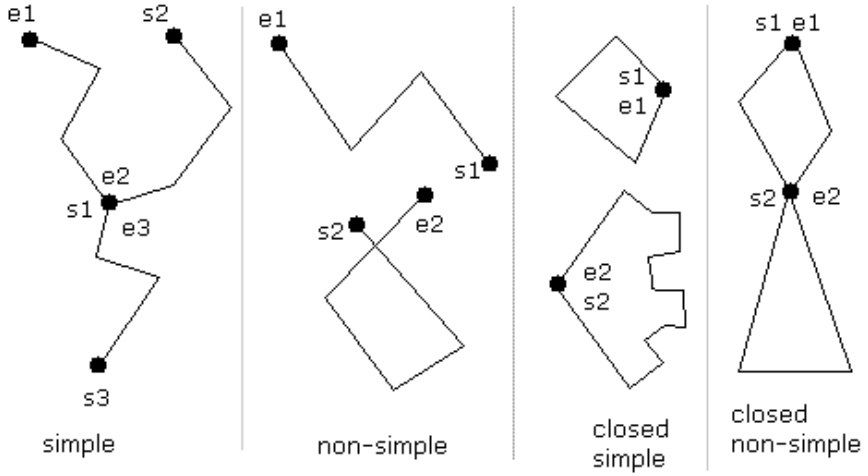
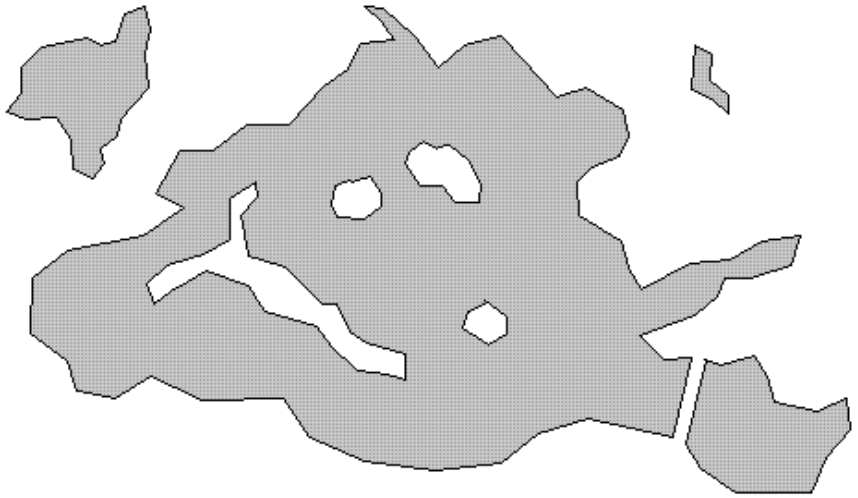
POLYGON



POLYGONs

MULTIPOINT

MULTIPOINT

MULTILINESTRING	 MULTILINESTRINGs
MULTIPOLYGON	 MULTIPOLYGON
GEOMETRYCOLLECTION	Any arbitrary collection of elementary sub-classes. Please note: for some odd reason this one seems to be the sub-class absolutely beloved by *inexperienced beginners*: all them are fond of **GEOMETRYCOLLECTION**: • **GEOMETRYCOLLECTION** isn't supported by the *Shapefile* format. • And this sub-class isn't generally supported by ordinary GIS sw (*viewers and so on*). So it's very rarely used in the real GIS professional world.

WKT and WKT notations

`Geometry` is a very complex data type: accordingly to this, **OGC-SFS** defines two alternative standard notations allowing to represent Geometry values:

- the **WKT** (*Well Known Text*) notation is intended to be ***user friendly*** (*not really so user friendly after all, but at least* ***human readable***).
- the **WKB** (*Well Known Binary*) notation on the other side is more intended for precise and accurate ***import/export/exchange*** of Geometries between different platforms.

Dimension: **XY** (**2D**) *the most oftenly used ...*

Geometry Type	WKT example
POINT	**POINT(123.45 543.21)**
LINESTRING	**LINESTRING(100.0 200.0, 201.5 102.5, 1234.56 123.89)** *three vertices*
POLYGON	**POLYGON((101.23 171.82, 201.32 101.5, 215.7 201.953, 101.23 171.82))** *exterior ring, no interior rings* --- **POLYGON((10 10, 20 10, 20 20, 10 20, 10 10),** **(13 13, 17 13, 17 17, 13 17, 13 13))** *exterior ring, one interior ring*
MULTIPOINT	**MULTIPOINT(1234.56 6543.21, 1 2, 3 4, 65.21 124.78)** *three points*
MULTILINESTRING	**MULTILINESTRING((1 2, 3 4), (5 6, 7 8, 9 10), (11 12, 13 14))** *first and last linestrings have 2 vertices each one;* *the second linestring has 3 vertices*
MULTIPOLYGON	**MULTIPOLYGON(((0 0,10 20,30 40,0 0),(1 1,2 2,3 3,1 1)),** **((100 100,110 110,120 120,100 100)))** *two polygons: the first one has an interior ring*
GEOMETRYCOLLECTION	**GEOMETRYCOLLECTION(POINT(1 1),** **LINESTRING(4 5, 6 7, 8 9),** **POINT(30 30))**

XYZ (**3D**)

Geometry Type	WKT example
POINT	**POINTZ(13.21 47.21 0.21)**
LINESTRING	**LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33)**
POLYGON	...
MULTIPOINT	**MULTIPOINTZ(15.21 57.58 0.31, 15.81 57.12 0.33)**

MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	**GEOMETRYCOLLECTIONZ(POINTZ(13.21 47.21 0.21), LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33))**

Dimension: **XYM** (**2D + M***easure*)
Please note: this one has nothing to do with 3D.
M is a **m***easure* value, not a geometry dimension.

Geometry Type	WKT example
POINT	**POINTM(13.21 47.21 1000.0)**
LINESTRING	**LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0)**
POLYGON	...
MULTIPOINT	**MULTIPOINTM(15.21 57.58 1000.0, 15.81 57.12 1100.0)**
MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	**GEOMETRYCOLLECTIONM(POINTM(13.21 47.21 1000.0), LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0))**

XYZM (**3D + M*easure***)
Please note: **M** is a **m***easure* value, not a geometry dimension.

Geometry Type	WKT example
POINT	**POINTZM(13.21 47.21 0.21 1000.0)**
LINESTRING	**LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)**
POLYGON	...
MULTIPOINT	**MULTIPOINTZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)**
MULTILINESTRING	...
MULTIPOLYGON	...
GEOMETRYCOLLECTION	**GEOMETRYCOLLECTIONZM(POINTZM(13.21 47.21 0.21 1000.0), LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0))**

There are several Spatial SQL functions supporting **WKT** and **WKB** handling;
examining all them one by one will surely be absolutely boring (*and not really useful for the average user*).
So we'll briefly explore just the main (*and most often used*) ones.

```
SELECT Hex(ST_GeomFromText('POINT(1.2345 2.3456)'));
```

```
0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE
```

```
SELECT
ST_AsText(x'0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE');
```

```
POINT(1.2345 2.3456)
```

```
SELECT
Hex(ST_AsBinary(x'0001FFFFFFFF8D976E1283C0F33F16FBCBEEC9C302408D976E1283C0F33F16FBCBEEC9C302407C010000008D976E1283C0F33F16FBCBEEC9C30240FE'));
```

```
01010000008D976E1283C0F33F16FBCBEEC9C30240
```

```
SELECT Hex(ST_AsBinary(ST_GeomFromText('POINT(1.2345 2.3456)')));
```

```
01010000008D976E1283C0F33F16FBCBEEC9C30240
```

```
SELECT ST_AsText(ST_GeomFromWKB(x'01010000008D976E1283C0F33F16FBCBEEC9C30240'));
```

```
POINT(1.2345 2.3456)
```

Please note well: both **WKT** and **WKB** notations are intended to support standard data exchange (import/export);
anyway the actual format internally used by SpatiaLite is a different one, i.e. **BLOB Geometry**.
You must never be concerned about such *internal format*:
you simply have to use the appropriate conversion functions so to convert back and forth in standard **WKT** or **WKB**.

- the **Hex()** function is a standard SQL function allowing to represent binary values as hexadecimal encoded text strings.
- the Spatial SQL function **ST_GeomFromText()** converts any valid **WKT** expression into an internal **BLOB Geometry** value.
- **ST_GeomFromWKB()** converts any valid **WKB** expression into an internal **BLOB Geometry** value.
- **ST_AsText()** converts an *internal* **BLOB Geometry** value into the corresponding **WKT** expression.
- **ST_AsBinary()** converts an *internal* **BLOB Geometry** value into the corresponding **WKB** expression.

```
SELECT ST_GeometryType(ST_GeomFromText('POINT(1.2345 2.3456)'));
```

```
POINT
```

```
SELECT ST_GeometryType(ST_GeomFromText('POINTZ(1.2345 2.3456 10)'));
```

```
POINT Z
```

```
SELECT ST_GeometryType(ST_GeomFromText('POINT ZM(1.2345 2.3456 10 20)'));
```

```
POINT ZM
```

ST_GeometryType() will return the Geometry Type from the given *internal* **BLOB Geometry** value.

- **Please note:** when using not-2D dimensions, declaring e.g. '**POINTZ**' or '**POINT Z**' is absolutely the same: SpatiaLite understands both notations indifferently.

```
SELECT ST_Srid(ST_GeomFromText('POINT(1.2345 2.3456)'));
-1

SELECT ST_Srid(ST_GeomFromText('POINT(1.2345 2.3456)', 4326));
4326
```

ST_Srid() will return the SRID from the given *internal* **BLOB Geometry**> value.

- **Please note:** both **ST_GeomFromText()** and **ST_GeomFromWKB()** accept an *optional* **SRID** argument.
  If the **SRID** is unspecified (*not at all a good practice*), then **-1** is assumed.

Common pitfalls

*"I've declared a **MULTIPOINT**-type Geometry column;*
*now I absolutely have to insert a simple **POINT** into this table,*
*but I get a **constraint failed** error ..."*

Any **MULTIxxxxx** type can store a single elementary item: you simply have to use the appropriate **WKT** syntax.
And anyway several useful ***type casting*** functions exist.

```
SELECT ST_GeometryType(ST_GeomFromText('MULTIPOINT(1.2345 2.3456)'));
MULTIPOINT

SELECT ST_AsText(CastToMultiLineString(ST_GeomFromText('LINESTRING(1.2345 2.3456, 12.3456 23.4567)')));
MULTILINESTRING((1.2345 2.3456, 12.3456 23.4567))

SELECT ST_AsText(CastToXYZM(ST_GeomFromText('POINT(1.2345 2.3456)')));
POINT ZM(1.2345 2.3456 0 0)

SELECT ST_AsText(CastToXY(ST_GeomFromText('POINT ZM(1.2345 2.3456 10 20)')));
POINT(1.2345 2.3456)
```

Spatial MetaData Tables

2011 January 28

SpatiaLite requires several ***metadata tables*** in order to work properly.
There is absolutely nothing strange in such tables; they simply are tables as any other one.
They are collectively as ***metadata*** because they are collectively intended to support an extended and complete qualification of Geometries.

Quite any Spatial SQL function strongly relies on such tables: so they are absolutely required for internal management purposes.
Any attempt to ***hack*** someway such tables will quite surely result in a severely corrupted (*and malfunctioning*) database.

There is a unique safe way to interact with ***metadata tables***, i.e. using as far as possible the appropriate Spatial SQL functions.
Directly performing **`INSERT`**, **`UPDATE`** or **`DELETE`** on their behalf is a completely **unsafe** and **strongly discouraged** practice.

```
SELECT InitSpatialMetaData();
```

The **`InitSpatialMetaData()`** function must be called immediately after creating a new database, and before attempting to call any other Spatial SQL function:

- the scope of this function is exactly the one to create (*and populate*) any *metadata table* internally required by SpatiaLite.
- if any *metadata table* already exist, this function doesn't apply any action at all:
  so, calling more times **`InitSpatialMetaData()`** is useless but completely harmless.
- **please note:** **`spatialite_gui`** will automatically perform any required initialization task every time a new database is created:
  so (*using this tool*) there is no need at all to explicitly call this function.

```
SELECT *
FROM spatial_ref_sys;
```

srid	auth_name	auth_srid	ref_sys_name	proj4text	srs_wkt
			Anguilla 1957 /	+proj=tmerc +lat_0=0 +lon_0=-62	PROJCS["Anguilla 1957 / British West Indies Grid", GEOGCS["Anguilla 1957", DATUM["Anguilla_1957", SPHEROID["Clarke 1880 (RGS)",6378249.145,293.465, AUTHORITY["EPSG","7012"]], AUTHORITY["EPSG","6600"]], PRIMEM["Greenwich",0, AUTHORITY["EPSG","8901"]], UNIT["degree",0.01745329251994328,

2000	epsg	2000	British West Indies Grid	+k=0.9995000000000001 +x_0=400000 +y_0=0 +ellps=clrk80 +units=m +no_defs	AUTHORITY["EPSG","9122"], AUTHORITY["EPSG","4600"], UNIT["metre",1, AUTHORITY["EPSG","9001"], PROJECTION["Transverse_Mercator"], PARAMETER["latitude_of_origin",0], PARAMETER["central_meridian",-62], PARAMETER["scale_factor",0.9995], PARAMETER["false_easting",400000], PARAMETER["false_northing",0], AUTHORITY["EPSG","2000"], AXIS["Easting",EAST], AXIS["Northing",NORTH]]
...	...	...	...	...	...

The **spatial_ref_sys** table does actually contains the whole **EPSG dataset** (*Spatial Reference System definitions*).

- the **SRID** column is the **PRIMARY KEY** uniquely identifying each item.
- the **auth_name**<, **auth_srid** and **ref_sys_name** columns usually contains a reference to the original EPSG definition (*mainly for documentation purposes*).
- the **proj4text** column contains **geodesic parameters** required by the PROJ.4 library.
  - these parameters are absolutely required by the **Transform()** function, because any coordinate re-projection will be actually performed invoking the appropriate **PROJ.4** functions.
- the **srs_wkt** column contains a complete definition of the corresponding **SRS** using the (*obnoxiously verbose*) **WKT** format.
  - SpatiaLite itself doesn't requires this information to be present: but if this **WKT** string is available, then a **.PRJ** file will be created when exporting any Shapefile
    (*many GIS packages require a **.PRJ** file to be present for each Shapefile*).
  - please, don't be confused: this **WKT** for SRS has nothing to do with the better known **WKT** used to represent geometries.
- **important notice:** altering the original EPSG definitions is unsafe and strongly discouraged, and must be absolutely avoided.
  Anyway you are absolutely free to insert further ***custom*** definitions by your own:
  in this case using SRID values **> 32768** is strongly suggested.

```
SELECT *
FROM geometry_columns;
```

f_table_name	f_geometry_column	type	coord_dimension	srid	spatial_index_enabled
local_councils	geometry	MULTIPOLYGON	XY	23032	1
populated_places	geometry	POINT	XY	4326	1

The **geometry_columns** table supports each Geometry column defined into the database:

- any column not supported by a corresponding entry within this table, for sure cannot be considered as a *genuine* Geometry.
- **important notice:** any attempt to ***hack*** this table by directly performing **INSERT**, **UPDATE** or **DELETE** will quite surely end into a ***major disaster***
  (*i.e. a corrupted and malfunctioning database*).
  Use the appropriate SQL functions instead: **AddGeometryColumn()**, **RecoverGeometryColumn()** and so on.

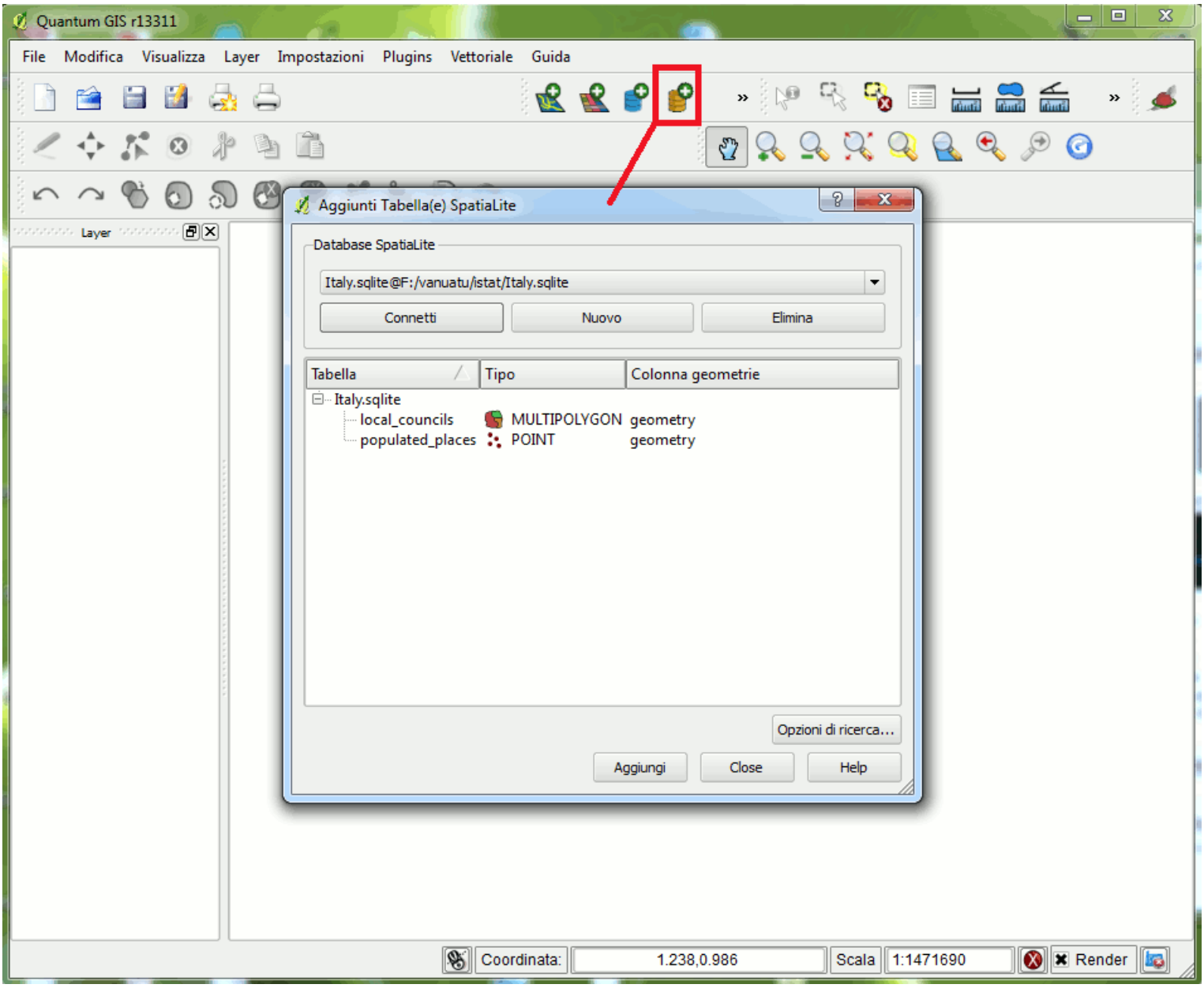
The **geometry_columns** table is intended to support *ordinary* tables.
Anyway two further similar tables exist as well:

- the **views_geometry_columns** is intended to support Geometry **VIEW**s.
- and the **virts_geometry_columns** is intended to support Virtual Shapefiles.

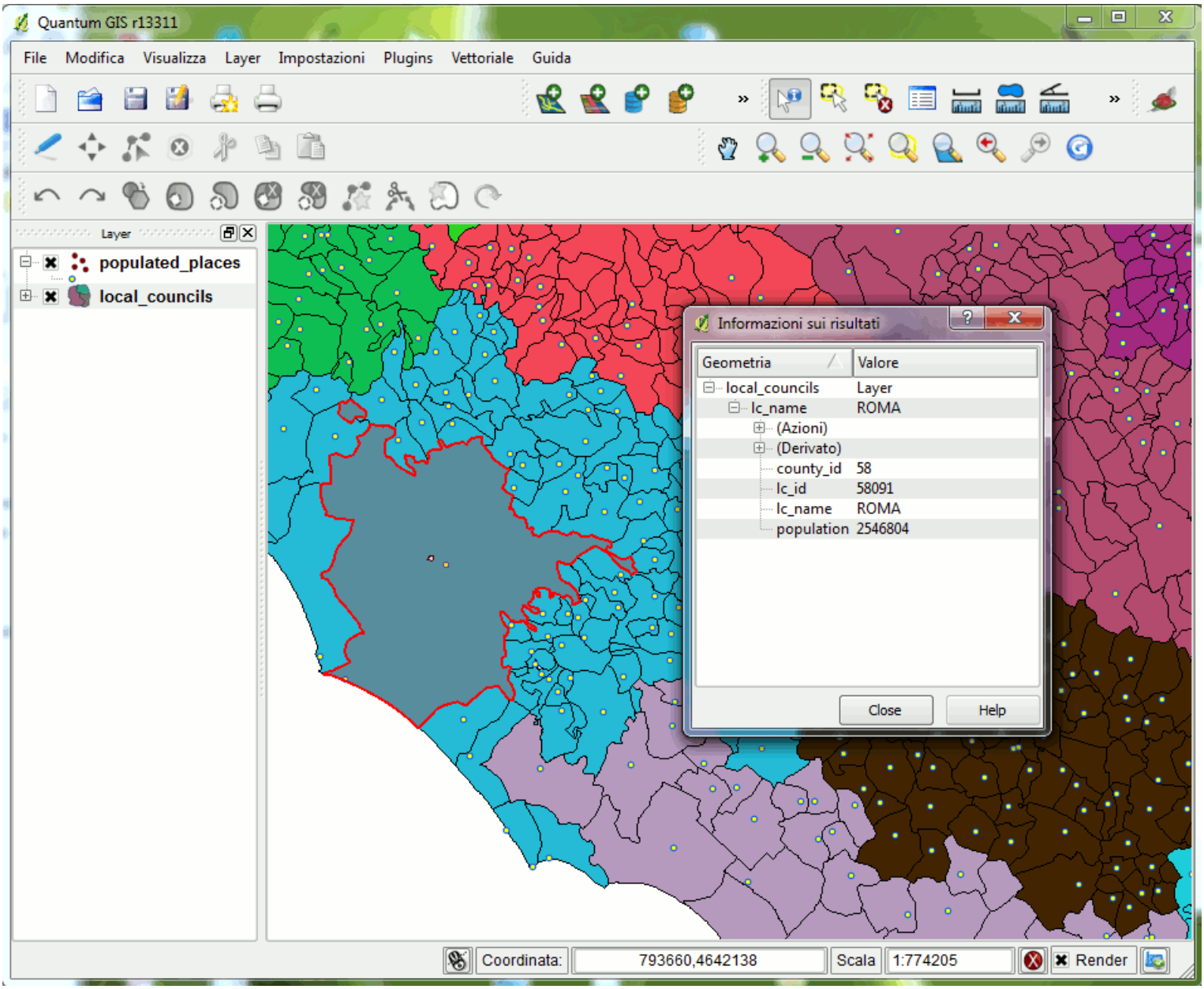
Viewing SpatiaLite layers in QGIS

2011 January 28

QGIS is a really popular and widespread desktop GIS app: you can download the latest QGIS from:
http://www.qgis.org/
QGIS contains an internal data provider supporting SpatiaLite:
so interacting with any SpatiaLite's DB using a *classic* desktop GIS is simple and easy.



You simply have to connect the SpatiaLite's DB, then choosing the layer(s) you intend to use.
Please note: accordingly to DBMS terminology you are accustomed to handle **tables**.
But in the GIS own jargon the term **layers** is very often used to identify exactly the same thing.



Once you've connected your layers from the SpatiaLite DB you can immediately start using QGIS own tools.
And that's all.

Recipe #1:
Creating a well designed DB

2011 January 28

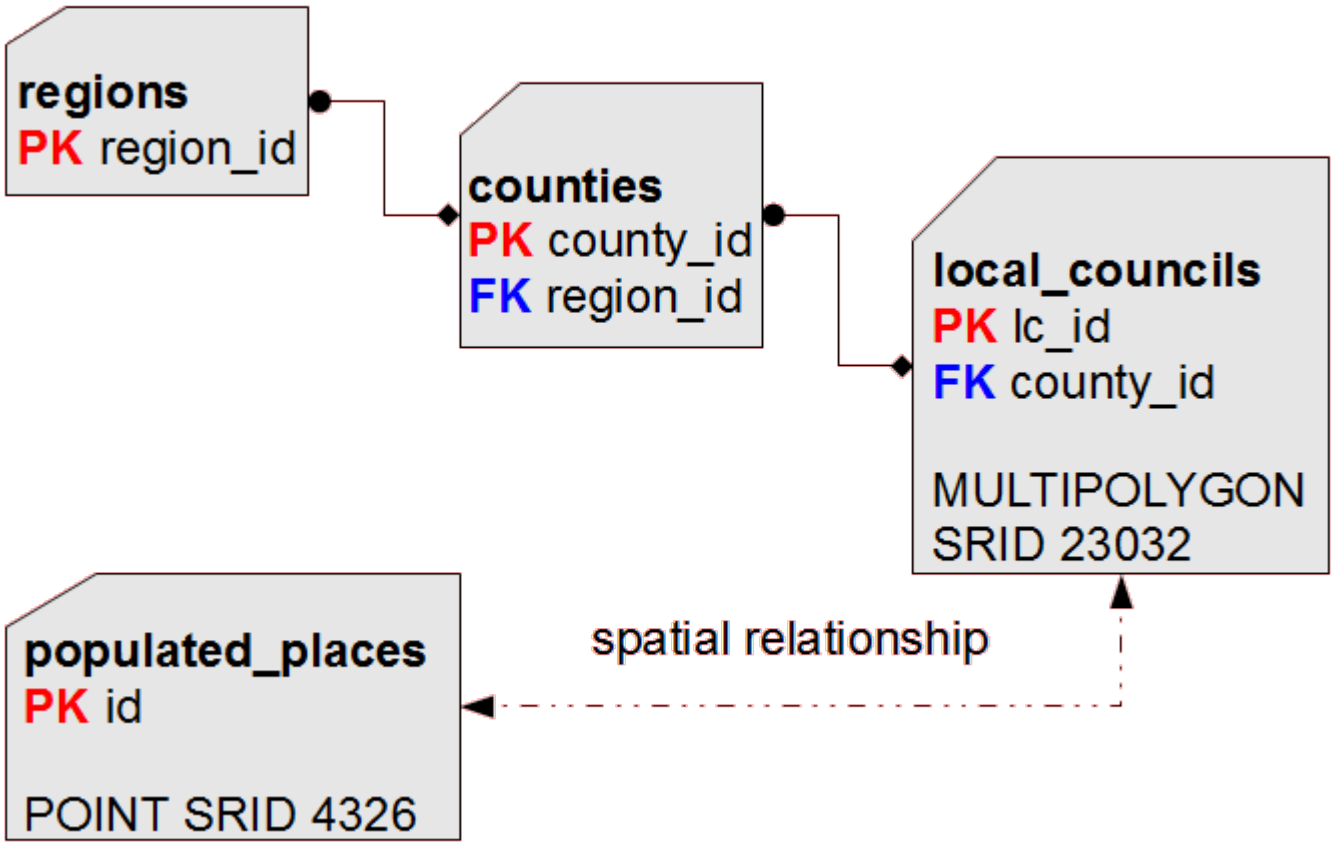
Normal Form

Any *well designed* DB adheres to the **relational** paradigm, and implements the so-called ***Normal Form***.
Very simply explained in plain words:

- you'll first attempt to identify any distinct **category** (aka class) present into your dataset
- and simultaneously you have to identify any possible **relation** connecting categories.
- data **redundancy** is strongly discouraged, and has to be reduced whenever is possible.

Consider the **ISTAT Census 2001**; identifying categories and relations is absolutely simple:

- At the lowermost hierarchy level we have obviously Local Councils.
- Each Local Council surely belongs to some County: so a relation exists connecting Local Councils and Counties.
  To be more descriptive, this one is a typical ***one-to-many*** relationship
  (one single County / many Local Councils: placing the same Local Council on two different Counties is absolutely forbidden).
- The same is true for Counties and Regions.
- There is not real need to establish a relation between Local Councils and Regions, because we can get this relation using the County as an intermediate pivot.



Accordingly to this, it's quite easy to identify several flaws in the original *Shapefile's* layout:

1. a **`POP2001`** value is present for Local Councils, Counties and Regions:
   well, this one clearly is an unneeded redundancy.
   We simply have to preserve this information at the lowermost level (Local Councils):
   because we can then compute anyway an aggregate value for Counties (or Regions).
2. a second redundancy exists: there is no real need compelling us to store both County and Region codes for each Local Council.
   Preserving the County code is just enough, because we can get a reference to the corresponding Region anyway simply referencing the County.
3. a Geometry representation is stored for each County and Region:
   this too represents an unneeded redundancy, because we can get such Geometries simply aggregating the ones stored at the Local Council level.

Then we have the **`cities1000`** dataset: which comes from a completely different source (so there is no useful key we can use to establish relations to other entities).
And this dataset is in the **`4326 SRID`** (**WGS84**), whilst any **ISTAT - Census 2001** dataset is in the **`23032 SRID`** [**ED50 UTM zone 32**];
so for now will simply keep this dataset in a completely self-standing state.
We'll see later how we can actually integrate this dataset with the other ones: after all, all them represent Italy, isn't ?
For sure some geographic relationship must exist ...

```
CREATE TABLE regions (
  region_id INTEGER NOT NULL PRIMARY KEY,
  region_name TEXT NOT NULL);
```

Step 1a) we'll start creating the **`regions`** table (i.e. the one positioned at the topmost hierarchic level).
Please note: we have defined a **`PRIMARY KEY`**, i.e. a unique (not duplicable), absolutely unambiguous identifier for each Region.

```
INSERT INTO regions (region_id, region_name)
SELECT COD_REG, REGIONE
```

```
FROM reg2001_s;
```

Step 1b) then we'll populate the **regions** table.
Using the **INSERT INTO ... SELECT ...** is more or less like performing a copy:
rows are extracted from the input table and immediately inserted into the output table.
As you can see, corresponding columns are explicitly identified *by order*.

```
CREATE TABLE counties (
  county_id INTEGER NOT NULL PRIMARY KEY,
  county_name TEXT NOT NULL,
  car_plate_code TEXT NOT NULL,
  region_id INTEGER NOT NULL,
  CONSTRAINT fk_county_region
    FOREIGN KEY (region_id)
    REFERENCES regions (region_id));
```

```
INSERT INTO counties (county_id, county_name,
  car_plate_code, region_id)
SELECT cod_pro, provincia, sigla, cod_reg
FROM prov2001_s;
```

Step 2a) we'll now create (and populate) the **counties** table.
Please note: a relation exists linking **counties** and **regions**.
Defining an appropriate **FOREIGN KEY** we'll make such relation to be explicitly set once for all.

```
CREATE INDEX idx_county_region
  ON counties (region_id);
```

Step 2b) accordingly to performance considerations, we must also create an INDEX corresponding to each **FOREIGN KEY** we'll define.

Very shortly explained: a **PRIMARY KEY** isn't simply a logical constraint.
In SQLite defining a **PRIMARY KEY** automatically implies generating an implicit index supporting fast direct access to each single row.
But on the other side defining a **FOREIGN KEY** simply establishes a logical constraint:
so if you actually wish to support fast direct access to each single row you have to explicitly create the corresponding index.

```
CREATE TABLE local_councils (
  lc_id INTEGER NOT NULL PRIMARY KEY,
  lc_name TEXT NOT NULL,
  population INTEGER NOT NULL,
  county_id INTEGER NOT NULL,
  CONSTRAINT fk_lc_county
    FOREIGN KEY (county_id)
    REFERENCES counties (county_id));
```

```
CREATE INDEX idx_lc_county
ON local_councils (county_id);
```

Step 3a) we'll now create the **local_councils** table.
A relation exists linking **local_councils** and **counties**.
So in this case too we have to define a **FOREIGN KEY** , then creating the corresponding index.
Please note: we haven't defined any Geometry column, although one is required for **local_councils**;
this is not a mistake, this is absolutely intentional.

```
SELECT AddGeometryColumn(
  'local_councils', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');
```

Step 3b) creating a Geometry column isn't the same as creating any other ordinary column.
We have to use the **AddGeometryColumn()** spatial function, specifying:

1. the **table** name

2. the **geometry** column name
3. the **SRID** to be used
4. the expected **geometry class**
5. the **dimension model**
   (in this case, simple 2D)

```
INSERT INTO local_councils (lc_id,
  lc_name, population, county_id, geometry)
SELECT PRO_COM, NOME_COM, POP2001,
  COD_PRO, Geometry
FROM com2001_s;
```

Step 3c) after all this can populate the **`local_councils`** table as usual.

```
CREATE TABLE populated_places (
  id INTEGER NOT NULL
    PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL);
```

```
SELECT AddGeometryColumn(
  'populated_places', 'geometry',
  4326, 'POINT', 'XY');
```

```
INSERT INTO populated_places (id,
  name, geometry)
SELECT NULL, COL002,
  MakePoint(COL006, COL005, 4326)
FROM cities1000
WHERE COL009 = 'IT';
```

Step 4) you have now to perform the last step: creating (and populating) the **`populated_places`** table.
Several interesting points to be noted:

- we have used an **`AUTOINCREMENT`** clause for the **`PRIMARY KEY`**
- this practically means that SQLite can automatically generate an appropriate unique value for this **`PRIMARY KEY`**, when no explicit value has been already set.
- accordingly to this, in the **`INSERT INTO`** statement a **`NULL`** value was set for the **`PRIMARY KEY`**:
  and this explicitly solicited SQLite to assign automatic values.
- the original **`cities1000`** dataset shipped two numeric columns for **longitude** [**`COL006`**] and **latitude** [**`COL005`**]:
  so we have to use the **`MakePoint()`** Spatial function in order to build a point-like Geometry.
- using the **`SRID 4326`** we set such Geometry into the **WGS84** [**Geographic System**] SRS.

Just to recapitulate:

- You started this tutorial using **Virtual Shapefiles** (and **Virtual CSV/TXT**) tables.
- Such **Virtual Tables** aren't at all real DB tables: they aren't *internally stored*.
  They simply are trivial *external files* accessed using an appropriate driver.
- Using Virtual Tables at first allowed you to test some simple and very basic SQL queries.
- But in order to test more complex SQL features any dataset have to be properly **imported** into the DBMS itself.
- And this step required creating (and then populating) ***internal tables***, accordingly to a well designed layout.

```
DROP TABLE com2001_s;
DROP TABLE prov2001_s;
DROP TABLE reg2001_s;
DROP TABLE cities1000;
```

Step 5) and finally you can drop any **Virtual Table**, because they aren't any longer useful.
Please note: dropping a *Virtual Shapefile* or *Virtual CSV/TXT* doesn't removes the corresponding *external* data-source, but simply removes the connection with the current database.

Recipe #2:
Your first JOIN queries

2011 January 28

You already know the basic foundations about *simple* SQL queries.
Any previous example encountered since now simply queried a single table:
anyway SQL has no imposed limits, so you can query an arbitrary number of tables at the same time.
But in order to do this you must understand how to correctly handle a **`JOIN`**.

```
SELECT *
FROM counties, regions;
```

county_id	county_name	car_plate_code	region_id	region_id	region_name
1	TORINO	TO	1	1	PIEMONTE
1	TORINO	TO	1	2	VALLE D'AOSTA
1	TORINO	TO	1	3	LOMBARDIA
1	TORINO	TO	1	4	TRENTINO-ALTO ADIGE
1	TORINO	TO	1	5	VENETO
...	...	...	...	...	...

Apparently this query immediately works;
but once you get a quick glance at the result-set you'll immediately discover something really puzzling:

- an unexpected huge number of rows has been returned.
- and each single County seems to be related with any possible Region.

Every time SQL queries two different tables at the same time, the **Cartesian Product** of both datasets is calculated.
i.e. each row coming from the first dataset is **`JOIN`**ed with any possible row coming from the second dataset.
This one is a ***blind*** combinatorial process, so it very difficultly can produce useful results.
And this process can easily generate a ***really huge result-set***: this must absolutely be avoided, because:

- a very long (*very*, *very long*) time may be required to complete the operation.
- you can easily exhaust operating system resources before completion.

All this said, it's quite obvious that some appropriate **JOIN condition** has to be set in order to maintain under control the Cartesian Product, so to actually return only meaningful rows.

```
SELECT *
FROM counties, regions
WHERE counties.region_id = regions.region_id;
```

This query is exactly the same of the previous one: but this time we introduced an appropriate **JOIN condition**.
Some points to be noted:

- using two (*or more*) tables can easily lead to *name ambiguity*:
  e.g. in this case we have two different columns named **region_id**, one in the **counties** table, the other in the **regions** table.
- we must use *fully qualified names* to avoid any possible ambiguity:
  e.g. **counties.region_id** identifies the **region_id** column belonging to the **counties** table, in an absolutely unambiguous way.
- defining the **WHERE counties.region_id = regions.region_id** clause we impose an appropriate ***JOIN condition***.
  After this the Cartesian Product will be accordingly filtered, so to insert into the result-set only the rows actually satisfying the imposed condition, ignoring any other.

```
SELECT c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name
FROM counties AS c,
  regions AS r
WHERE c.region_id = r.region_id;
```

county_id	county_name	car_plate_code	region_id	region_name
1	TORINO	TO	1	PIEMONTE
2	VERCELLI	VC	1	PIEMONTE
3	NOVARA	NO	1	PIEMONTE
4	CUNEO	CN	1	PIEMONTE
5	ASTI	AT	1	PIEMONTE
6	ALESSANDRIA	AL	1	PIEMONTE
...	...	...	...	...

And this one always is the same as above, simply written adopting a most polite syntax:

- using extensively the **AS** clause so to define alias names for both columns and tables make **JOIN** queries to be much more concise and readable, and easiest to understand.

```
SELECT lc.lc_id AS lc_id,
  lc.lc_name AS lc_name,
  lc.population AS population,
  c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name
FROM local_councils AS lc,
  counties AS c,
  regions AS r
WHERE lc.county_id = c.county_id
  AND c.region_id = r.region_id;
```

lc_id	lc_name	population	county_id	county_name	car_plate_code	region_id	region_name
1001	AGLIE'	2574	1	TORINO	TO	1	PIEMONTE
1002	AIRASCA	3554	1	TORINO	TO	1	PIEMONTE
1003	ALA DI STURA	479	1	TORINO	TO	1	PIEMONTE
...	...	...	...	...	...	...	...

Joining three (*or even more*) tables isn't much more difficult:

you simply have to apply any required **JOIN condition** as appropriate.

Performance considerations

Executing complex queries involving many different tables may easily run in a very slow and sluggish mode.
This will most easily noticed when such tables contain a huge number of rows.
Explaining all this isn't at all difficult: in order to calculate the Cartesian Product the SQL engine has to access many and many times each table involved in the query.

The basic behavior is the one to perform a **full table scan** each time: and obviously scanning a long table many and many times requires a long time.
So the main key-point in order optimize your queries is the one to avoid using *full table scans* as much as possible.
All this is fully supported, and it's easy to be implemented.

Each time the ***SQL-planner*** (an internal component of the ***SQL-engine***) detects that an appropriate **`INDEX`** is available, there is no need at all to perform *full table scans*, because each single row can now be immediately accessed using this Index.
And this one will obviously be a much faster process.
Any column (or group of columns) frequently used in **`JOIN`** clauses is a good candidate for a corresponding **`INDEX`**.
Anyway, creating an Index implies several negative consequences:

- the storage allocation required by the DB-file will increase (sometimes will dramatically increase).
- performing **`INSERT`**, **`UPDATE`** and/or **`DELETE`** ops will require a longer time, because the Index has to be accordingly updated.
  And this obviously imposes a further overhead.

So (not surprisingly) it's a *trade-off process*: you must evaluate carefully when an **`INDEX`** is absolutely required, and attempt to reach a well balanced mix.
i.e a *compromise* between contrasting requirements, under various conditions and in different users-cases.
In other words there is no *absolute rule*: you must find your optimal *case-by-case* solution performing several practical tests, until you get the optimal solution fulfilling your requirements.

Recipe #3:
More about JOIN

2011 January 28

SQL supports another alternative syntax to represent JOIN ops.
More or less both implementations are strictly equivalent, so using the one or the other simply is matter of personal taster in the majority of cases.
Anyway, this second method supports some really interesting further feature that is otherwise unavailable.

```
SELECT lc.lc_id AS lc_id,
  lc.lc_name AS lc_name,
  lc.population AS population,
  c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name
FROM local_councils AS lc,
  counties AS c,
  regions AS r
WHERE lc.county_id = c.county_id
  AND c.region_id = r.region_id;
```

You now feel a strong ***deja vu*** sensation: and that's more than appropriate, because you have already encountered this query in the previous example.

```
SELECT lc.lc_id AS lc_id,
  lc.lc_name AS lc_name,
  lc.population AS population,
  c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name
FROM local_councils AS lc
JOIN counties AS c ON (
  lc.county_id = c.county_id)
JOIN regions AS r ON (
  c.region_id = r.region_id);
```

All right, this one is the same identical query rewritten accordingly to alternative syntax rules:

- using the **`JOIN ... ON (...)`** clause makes more explicit what's going on.
- and **`JOIN`** conditions are now directly expressed within the **`ON (...)`** term:
  this way the query statement is better structured and more readable.
- anyway, all this simply is ***syntactic sugar***:
  there is no difference at all between the above two queries in ***functional*** terms.

```
SELECT r.region_name AS region,
  c.county_name AS county,
  lc.lc_name AS local_council,
  lc.population AS population
FROM regions AS r
JOIN counties AS c ON (
```

```
  c.region_id = r.region_id)
JOIN local_councils AS lc ON (
  c.county_id = lc.county_id
  AND lc.population > 100000)
ORDER BY r.region_name,
  county_name;
```

region	county	local_council	population
ABRUZZO	PESCARA	PESCARA	116286
CALABRIA	REGGIO DI CALABRIA	REGGIO DI CALABRIA	180353
CAMPANIA	NAPOLI	NAPOLI	1004500
CAMPANIA	SALERNO	SALERNO	138188
EMILIA-ROMAGNA	BOLOGNA	BOLOGNA	371217
...	...	...	...

There is nothing strange or new in this query:

- we simply introduced a further **`ON (... AND lc.population < 100000)`** clause, so to exclude any small populated Local Council.

```
SELECT r.region_name AS region,
  c.county_name AS county,
  lc.lc_name AS local_council,
  lc.population AS population
FROM regions AS r
JOIN counties AS c ON (
  c.region_id = r.region_id)
LEFT JOIN local_councils AS lc ON (
  c.county_id = lc.county_id
  AND lc.population > 100000)
ORDER BY r.region_name,
  county_name;
```

region	county	local_council	population
ABRUZZO	CHIETI	NULL	NULL
ABRUZZO	L'AQUILA	NULL	NULL
ABRUZZO	PESCARA	PESCARA	116286
ABRUZZO	TERAMO	NULL	NULL
BASILICATA	MATERA	NULL	NULL
BASILICATA	POTENZA	NULL	NULL
...	...	...	...

Apparently this query is the same as the latest one.
But a remarkable difference exists:

- this time we've used a **`LEFT JOIN`** clause:
  and the result-set now looks very different from the previous one.
- a plain **`JOIN`** clause will include into the result-set only rows for which both the ***left-sided*** and the ***right-sided*** terms has been positively resolved.
- but the most sophisticated **`LEFT JOIN`** clause will include into the result-set any row having an unresolved ***right-sided*** term:
  and in this case any ***right-sided*** member assumes a **`NULL`** value.

There is a striking difference between a plain **`JOIN`** and a **`LEFT JOIN`**.
Coming back to previous example, using a **`LEFT JOIN`** clause ensures that any Region and any County will now be inserted into the result-set, even the ones failing to satisfy the imposed population limit for Local Councils.

Recipe #4: About VIEW

2011 January 28

SQL supports a really useful feature, the so called **VIEW**.
Very shortly explained, a **VIEW** is something falling half-way between a **TABLE** and a **query**:

- a **VIEW** is a persistent objects (exactly as **TABLE**s are).
- you can query a **VIEW** exactly in the same way you can query a **TABLE**:
  there is no difference at all distinguishing a **VIEW** and a **TABLE** from the **SELECT** own perspective.
- but after all a **VIEW** simply is like a kind of *glorified* query.
  A **VIEW** has absolutely no data by itself.
  Data *apparently* belonging to some **VIEW** are simply retrieved from some other **TABLE** each time they are actually required.
- in the SQLite's own implementation any **VIEW** strictly is a **read-only** object:
  you can freely reference any **VIEW** in **SELECT** statements.
  But attempting to perform an **INSERT**, **UPDATE** or **DELETE** statement on behalf of a **VIEW** isn't allowed.

Anyway, performing some practical exercise surely is the best way to introduce Views.

```
CREATE VIEW view_lc AS
SELECT lc.lc_id AS lc_id,
  lc.lc_name AS lc_name,
  lc.population AS population,
  c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name,
  lc.geometry AS geometry
FROM local_councils AS lc
JOIN counties AS c ON (
  lc.county_id = c.county_id)
JOIN regions AS r ON (
  c.region_id = r.region_id);
```

Et voila, here is your first **VIEW**:

- basically, this looks exactly as any **query** you've already seen since now.
- except in this; this time the first line is: **CREATE VIEW ... AS**
- and this one is unique syntactical difference transforming a plain query into a **VIEW** definition.

```
SELECT lc_name, population, county_name
FROM view_lc
WHERE region_name = 'LAZIO'
ORDER BY lc_name;
```

lc_name	population	county_name
ACCUMOLI	724	RIETI
ACQUAFONDATA	316	FROSINONE

ACQUAPENDENTE	5788	VITERBO
ACUTO	1857	FROSINONE
AFFILE	1644	ROMA
...	...	...

You can actually query this **VIEW**.

```
SELECT region_name,
  Sum(population) AS population,
  (Sum(ST_Area(geometry)) / 1000000.0)
    AS "area (sq.Km)",
  (Sum(population) /
    (Sum(ST_Area(geometry)) / 1000000.0))
      AS "popDensity (peoples/sq.Km)"
FROM view_lc
GROUP BY region_id
ORDER BY 4;
```

region_name	population	area (sq.Km)	popDensity (peoples/sq.Km)
VALLE D'AOSTA	119548	3258.405868	36.689107
BASILICATA	597768	10070.896921	59.355984
...	...	...	...
MARCHE	1470581	9729.862860	151.140979
TOSCANA	3497806	22956.355019	152.367656
...	...	...	...
LOMBARDIA	9032554	23866,529331	378.461144
CAMPANIA	5701931	13666.322146	417.224981

You can really perform any arbitrary complex query using a **VIEW**.

```
SELECT v.lc_name AS LocalCouncil,
  v.county_name AS County,
  v.region_name AS Region
FROM view_lc AS v
JOIN local_councils AS lc ON (
  lc.lc_name = 'NORCIA'
    AND ST_Touches(v.geometry, lc.geometry))
ORDER BY v.lc_name, v.county_name, v.region_name;
```

LocalCouncil	County	Region
ACCUMOLI	RIETI	LAZIO
ARQUATA DEL TRONTO	ASCOLI PICENO	MARCHE
CASCIA	PERUGIA	UMBRIA
CASTELSANTANGELO SUL NERA	MACERATA	MARCHE
CERRETO DI SPOLETO	PERUGIA	UMBRIA
CITTAREALE	RIETI	LAZIO
MONTEMONACO	ASCOLI PICENO	MARCHE
PRECI	PERUGIA	UMBRIA

You can **JOIN** a **VIEW** and a **TABLE** (or two **VIEW**s, and so on ...)
Just a simple explanation: this **JOIN** actually is one based on Spatial relationships:
the result-set represents the list of Local Councils sharing a common boundary with the **Norcia** one.
You can get a much more complete example here (*Haute cuisine* recipes).

VIEW is one of the many powerful and wonderful features supported by SQL.
And SQLite's own implementation for **`VIEW`** surely is a first class one.
You should use **`VIEW`** as often as you can: and you'll soon discover that following this way handling really complex DB layouts will become a piece of cake.

Please note: querying a **`VIEW`** can actually be as fast and efficient as querying a **`TABLE`**.
But a **`VIEW`** cannot anyway be more efficient than the underlying query is; any poorly designed and badly optimized query surely will translate into a very slow **`VIEW`**.

Recipe #5: Creating a new table (and related paraphernalia)

2011 January 28

You are now well conscious that SQL overall performance and efficiency strongly depend on the underlying **database layout**, i.e. the following design choices are critical:

- defining **tables** (and **columns**) in the most appropriate way.
- identifying **relations** connection different tables.
- supporting often-used relations with an appropriate **index**.
- identifying useful **constraints**, so to preserve data consistency and correctness as much as possible.

It's now time to examine in deeper detail such topics.

Pedantic note: in **DBMS/SQL** own jargon all this is collectively defined as **DDL** [*Data Definition Language*], and is intended as opposed to **DML** [*Data Manipulation Language*], i.e. **SELECT**, **INSERT** and so on.

```
CREATE TABLE peoples (
  first_name TEXT,
  last_name TEXT,
  age INTEGER,
  gender TEXT,
  phone TEXT);
```

This statement will create a very simple **table** named **peoples**:

- each individual **column** definition must at least specify the corresponding **data-type**, such as **TEXT** or **INTEGER**
- please note: data-type handling in SQLite strongly differs from others DMBS implementations: but we'll see this in more detail later.

```
CREATE TABLE peoples2 (
  id INTEGER NOT NULL
    PRIMARY KEY AUTOINCREMENT,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  age INTEGER
    CONSTRAINT age_verify
      CHECK (age BETWEEN 18 AND 90),
  gender TEXT
    CONSTRAINT gender_verify
      CHECK (gender IN ('M', 'F')),
  phone TEXT);
```

This one is more sophisticated version of the same table:

- we have added an **id** column, declared as **PRIMARY KEY AUTOINCREMENT**
  - inserting a **PRIMARY KEY** on each table always is a very good idea.
  - declaring an **AUTOINCREMENT** clause we'll request SQLite to automatically generate unique values for

this key.
- we have added a **`NOT NULL`** clause for **`first_name`** and **`last_name`** columns.
  - this will declare a first kind of **constraint**: **`NULL`** values will be rejected for these columns.
  - in other words, **`first_name`** and **`last_name`** absolutely have to contain some explicitly set value.
- we have added a **`CONSTRAINT ... CHECK (...)`** for **`age`** and **`gender`** columns.
  - this will declare a second kind of **constraint**: values failing to satisfy the **`CHECK (...)`** criterion will be rejected.
  - the **`age`** column will now accept only reasonable *adult* ages.
  - and the **`gender`** column will now accept only **`'M'`** or **`'F'`** values.
  - please note: we have not declared the **`NOT NULL`** clause, so **`age = NULL`** and **`gender = NULL`** are still considered to be valid values.

about SQLite data-types

Very shortly said: SQLite hasn't data-types at all ...
You are absolutely free to insert any data-type on every column: the column declared data-type simply have a *decorative* role, but isn't neither checked not enforced at all.
This one absolutely is not a **bug**: it's more a **peculiar design choice**.
Anyway, any other different DBMS applies strong data-type qualification and enforcement, so the SQLite's own behavior may easily look odd and puzzling. **Be warned**.

Anyway SQLite internally supports the following data-types:

- **`NULL`**: no value at all.
- **`INTEGER`**: actually **64bit** integers, so to support really huge values.
- **`DOUBLE`**: floating point, double precision.
- **`TEXT`**: any **UTF-8** encoded text string, of unconstrained arbitrary length.
- **`BLOB`**: any generic *Binary Long Object*, of unconstrained arbitrary length.

Remember: each single *cell* (*row/column* intersection) can store any arbitrary data-type.
One unique exception exists: columns declared as **`INTEGER PRIMARY KEY`** absolutely require integer values.

```
ALTER TABLE peoples2
  ADD COLUMN cell_phone TEXT;
```

You can add any further column even after the initial table creation.

Yet another SQLite's own very **peculiar design choice**.

- dropping columns is **unsupported**.
- renaming columns is **unsupported**.

i.e. once you've created a column there is no way at all to change its initial definition.

```
ALTER TABLE peoples2
  RENAME TO peoples_ok;
```

Anyway you are absolutely free to change the table name.

```
DROP TABLE peoples;
```

And this will completely erase the table (and its whole content) from the DB.

```
CREATE INDEX idx_peoples_phone
  ON peoples_ok (phone);
```

This will create an Index.

```
DROP INDEX idx_peoples_phone;
```

And this will destroy the same Index.

```
CREATE UNIQUE INDEX idx_peoples_name
  ON peoples_ok (last_name, first_name);
```

- an Index can support more columns.
- declaring the **`UNIQUE`** clause implements a further **constraint**:
  once some value is already defined, any further attempt to insert the same value will then fail.

```
PRAGMA table_info(peoples_ok);
```

cid	name	type	notnull	dflt_value	pk
0	id	INTEGER	1	NULL	1
1	first_name	TEXT	1	NULL	0
2	last_name	TEXT	1	NULL	0
3	age	INTEGER	0	NULL	0
4	gender	TEXT	0	NULL	0
5	phone	TEXT	0	NULL	0
6	cell_phone	TEXT	0	NULL	0

You can use **`PRAGMA table_info(...)`** in order to query a table layout.

```
PRAGMA index_list(peoples_ok);
```

seq	name	unique
0	idx_peoples_phone	0
1	idx_peoples_name	1

```
PRAGMA index_info(idx_peoples_name);
```

seqno	cid	name
0	2	last_name
1	1	first_name

And using **`PRAGMA index_list(...)`** and **`PRAGMA index_info(...)`** you can easily query the corresponding Index layout.

Recipe #6: Creating a new Geometry column

2011 January 28

We'll now examine in deeper detail how to correctly define a Geometry-type column.
SpatiaLite follows an approach very closely related to the one adopted by **PostgreSQL/PostGIS**;
i.e. creating a Geometry-type at the same time the corresponding table is created isn't allowed.
You always must first create the table, then adding the Geometry-column in a second time and as a separate step.

```
CREATE TABLE test_geom (
  id INTEGER NOT NULL
    PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  measured_value DOUBLE NOT NULL);
```

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
  4326, 'POINT', 'XY');
```

This is the only supported way to get a completely valid Geometry.
Any different approach will surely produce an incorrect and unreliable Geometry.

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
  4326, 'POINT', 'XY', 0);
```

```
SELECT AddGeometryColumn('test_geom', 'the_geom',
  4326, 'POINT', 'XY', 1);
```

Although the previous one surely is the most often used form, this one the complete form supported by **`AddGeometryColumn()`**:

- the last (*optional*) arguments actually means: **`NOT NULL`**
- setting a value **`ZERO`** (*assumed to be the default value if omitted*) then the Geometry column will accept **`NULL`** values.
- otherwise only **`NOT NULL`** geometries will be accepted.

Supported **SRID**s:

- any possible SRID defined within the **`spatial_ref_sys`** *metadata* table.
- or **`-1`** to politely denote any unknown / unspecified SRS.

Supported **Geometry-type**s:

Geometry Type	Notes
POINT	the commonly used Geometry-types: corresponding to Shapefile's specs supported by any desktop GIS apps
LINESTRING	
POLYGON	
MULTIPOINT	

MULTILINESTRING	
MULTIPOLYGON	
GEOMETRYCOLLECTION	Not often used: unsupported by Shapefile and desktop GIS apps
GEOMETRY	A generic container supporting any possible geometry-class Not often used: unsupported by Shapefile and desktop GIS apps

Supported **Dimension-model**s:

Dimension model	Alias	Notes
XY	2	X and Y coords (simple 2D)
XYZ	3	X, Y and Z coords (3D)
XYM		X and Y coords + *measure value* M
XYZM		X, Y and Z coords + *measure value* M

Please note well: this one is a very frequent pitfall.
Many developers, GIS professionals and alike obviously feel to be much smarter than this, so they often tend to invent some highly imaginative alternative way to create their own Geometries.
e.g. bungling someway the **`geometry_columns`** table seems to be a very popular practice.

May well be that such *creative* methods will actually work with some very specific SpatiaLite's version; but for sure some severe incompatibility will raise before or after ...

Be warned: only Geometries created using **`AddGeometryColumn()`** are fully legitimate.
Any different approach is completely unsafe (*and unsupported* ..)

I suppose that directly checking how AddGeometryColumn() affects the database may help you to understand better.

```
PRAGMA table_info(test_geom);
```

cid	name	type	notnull	dflt_value	pk
0	id	INTEGER	1	NULL	1
1	name	TEXT	1	NULL	0
2	measured_value	DOUBLE	1	NULL	0
3	the_geom	POINT	0	NULL	0

step 1: a new **`test_geom`** column has been added to the corresponding table.

```
SELECT *
FROM geometry_columns
WHERE f_table_name LIKE 'test_geom';
```

f_table_name	f_geometry_column	type	coord_dimension	srid	spatial_index_enabled
test_geom	the_geom	POINT	XY	4326	0

step 2: a corresponding row has been inserted into the **`geometry_columns`** metadata table.

```
SELECT *
FROM sqlite_master
WHERE type = 'trigger'
  AND tbl_name LIKE 'test_geom';
```

type	name	tbl_name	rootpage	sql
trigger	ggi_test_geom_the_geom	test_geom	0	CREATE TRIGGER "ggi_test_geom_the_geom" BEFORE INSERT ON "test_geom" FOR EACH ROW BEGIN SELECT RAISE(ROLLBACK, 'test_geom.the_geom violates Geometry constraint [geom-type or SRID not allowed]') WHERE (SELECT type FROM geometry_columns WHERE f_table_name = 'test_geom' AND f_geometry_column = 'the_geom' AND GeometryConstraints(NEW."the_geom", type, srid, 'XY') = 1) IS NULL; END
trigger	ggu_test_geom_the_geom	test_geom	0	CREATE TRIGGER "ggu_test_geom_the_geom" BEFORE UPDATE ON "test_geom" FOR EACH ROW BEGIN SELECT RAISE(ROLLBACK, 'test_geom.the_geom violates Geometry constraint [geom-type or SRID not allowed]') WHERE (SELECT type FROM geometry_columns WHERE f_table_name = 'test_geom' AND f_geometry_column = 'the_geom' AND GeometryConstraints(NEW."the_geom", type, srid, 'XY') = 1) IS NULL; END

step 3: the **`sqlite_master`** is the main *metadata* table used by SQLite to store internal objects.
As you can easily notice, each Geometry requires some **triggers** to be fully supported and well integrated into the DBMS workflow.
Not at all surprisingly, all this has to be defined in a strongly self-consistent way in order to let SpatiaLite work as expected.
If some element is missing or badly defined, the obvious consequence will be a defective and unreliable Spatial DBMS.

```
SELECT DiscardGeometryColumn('test_geom', 'the_geom');
```

This will remove any **metadata** and any **trigger** related to the given Geometry.
Please note: anyway this will leave any geometry-value stored within the corresponding table absolutely untouched.
Simply, after calling **`DiscardGeometryColumn(...)`** they aren't any longer fully qualified geometries, but anonymous and generic BLOB values.

```
SELECT RecoverGeometryColumn('test_geom', 'the_geom',
  4326, 'POINT', 'XY');
```

This will attempt to recreate any **metadata** and any **trigger** related to the given Geometry.
If the operation successfully completes, then the Geometry column is fully qualified.
In other words, there is absolutely no difference between a Geometry created by **`AddGeometryColumn()`** and another created by **`RecoverGeometryColumn()`**.
Very simply explained:

- **`AddGeometryColumn()`** is intended to create a new, empty column.
- **`RecoverGeometryColumn()`** is intended to recover in a second time an already existing (*and populated*) column.

Compatibility issues between different versions

SpatiaLite isn't eternally immutable.
Like any other human artifact and any other software package SpatiaLite tends to evolve during the time; and SQLite as well evolves during the time.

Solemn commitment: you are absolutely granted that any database-file generated by some previous (***older***) version can be safely operated using any later (***newer***) version of both SQLite and SpatiaLite.

Please note well: the opposite isn't necessarily true.
Attempting to operate a database-file generated by a most recent (***newer***) version using any previous (***older***) version may easily be impossible at all, or may cause some more or less serious trouble.

Sometimes circumventing version-related issues is inherently impossible: e.g. there is absolutely no way to use 3D geometries on obsolescent versions, because the required support was introduced in more recent times.
But in many other cases such issues are simply caused by some incompatible binary function required by **triggers**.

Useful hint

To resolve any trigger-related incompatibility you can simply try to:

- remove first any trigger: the best way you can follow is using **`DiscardGeometryColumn()`**
- and then recreate again the triggers using **`AddGeometryColumn()`**

This will ensure that any *metadata info* and *trigger* will surely match expectations of your binary library current version.

Recipe #7:
Insert, Update and Delete

2011 January 28

Since now we've mainly examined how to query tables.
SQL isn't obviously a *read-only* language: inserting new rows, deleting existing rows and updating values is supported in the most flexible way.

It's now time to examine such topics in deeper detail.

```
CREATE TABLE test_geom (
  id INTEGER NOT NULL
    PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  measured_value DOUBLE NOT NULL);

SELECT AddGeometryColumn('test_geom', 'the_geom',
  4326, 'POINT', 'XY');
```

Nothing new in this: it's exactly the same table we've already created in the previous example.

```
INSERT INTO test_geom
    (id, name, measured_value, the_geom)
  VALUES (NULL, 'first point', 1.23456,
    GeomFromText('POINT(1.01 2.02)', 4326));

INSERT INTO test_geom
  VALUES (NULL, 'second point', 2.34567,
    GeomFromText('POINT(2.02 3.03)', 4326));

INSERT INTO test_geom
    (id, name, measured_value, the_geom)
  VALUES (10, 'tenth point', 10.123456789,
    GeomFromText ('POINT(10.01 10.02)', 4326));

INSERT INTO test_geom
    (the_geom, measured_value, name, id)
  VALUES (GeomFromText('POINT(11.01 11.02)', 4326),
    11.123456789, 'eleventh point', NULL);

INSERT INTO test_geom
    (id, measured_value, the_geom, name)
  VALUES (NULL, 12.123456789, NULL, 'twelfth point');
```

The **`INSERT INTO (...) VALUES (...)`** statement does exactly what its name states:

- the first list enumerates the column names
- and the second list contains the values to be inserted: correspondences between columns and values are established *by position*.
- you can simply suppress the column-names list (*please, see the second* **`INSERT`** *statement*):
  anyway this one isn't a good practice, because such statement implicitly relies upon some *default* column order, and that's not at all a safe assumption.
- another interesting point; this table declares a **`PRIMARY KEY AUTOINCREMENT`**:
  - as a general rule we've passed a corresponding **`NULL`** value, so to allow SQLite autogenerating a unique value.
  - but in the third **`INSERT`** we've set an explicit value (*please, check in the following paragraph what*

really happens in this case).

```
SELECT *
FROM test_geom;
```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
2	second point	2.345670	BLOB sz=60 GEOMETRY
10	tenth point	10.123457	BLOB sz=60 GEOMETRY
11	eleventh point	11.123457	BLOB sz=60 GEOMETRY
12	twelfth point	12.123457	NULL

Just a quick check before going further on ...

```
INSERT INTO test_geom
  VALUES (2, 'POINT #2', 2.2,
    GeomFromText('POINT(2.22 3.33)', 4326));
```

This further **INSERT** will ***loudly*** fail, raising a **constraint failed** exception.
Accounting for this isn't too much difficult: a **PRIMARY KEY** always enforces a ***uniqueness constraint***.
And actually one row of **id = 2** already exists into this table.

```
INSERT OR IGNORE INTO test_geom
  VALUES (2, 'POINT #2', 2.2,
    GeomFromText('POINT(2.22 3.33)', 4326));
```

By specifying an **OR IGNORE** clause this statement will now ***silently*** fail (*same reason as before*).

```
INSERT OR REPLACE INTO test_geom
  VALUES (2, 'POINT #2', 2.2,
    GeomFromText('POINT(2.22 3.33)', 4326));
```

There is a further variant: i.e. specifying an **OR REPLACE** clause this statement will actually act like an **UPDATE**

```
REPLACE INTO test_geom
    (id, name, measured_value, the_geom)
  VALUES (3, 'POINT #3', 3.3,
    GeomFromText('POINT(3.33 4.44)', 4326));

REPLACE INTO test_geom
    (id, name, measured_value, the_geom)
  VALUES (11, 'POINT #11', 11.11,
    GeomFromText('POINT(11.33 11.44)', 4326));
```

And yet another syntactic alternative is supported, i.e. simply using **REPLACE INTO**:
but this latter simply is an *alias* for **INSERT OR REPLACE**.

```
SELECT *
FROM test_geom;
```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
2	POINT #2	2.200000	BLOB sz=60 GEOMETRY
3	POINT #3	3.300000	BLOB sz=60 GEOMETRY
10	tenth point	10.123457	BLOB sz=60 GEOMETRY
11	POINT #11	11.110000	BLOB sz=60 GEOMETRY
12	twelfth point	12.123457	NULL

Just another quick check ...

```
UPDATE test_geom SET
  name = 'point-3',
  measured_value = 0.003
WHERE id = 3;

UPDATE test_geom SET
  measured_value = measured_value + 1000000.0
WHERE id > 10;
```

updating values isn't much more complex ...

```
DELETE FROM test_geom
WHERE (id % 2) = 0;
```

and the same is for deleting rows.
i.e. this **DELETE** statement will affect every *even* **id** value.

```
SELECT *
FROM test_geom;
```

id	name	measured_value	the_geom
1	first point	1.234560	BLOB sz=60 GEOMETRY
3	point-3	0.003000	BLOB sz=60 GEOMETRY
11	POINT #11	1000011.110000	BLOB sz=60 GEOMETRY

A last final quick check ...

Very important notice

Be warned: calling an **UPDATE** or **DELETE** statement without specifying any corresponding **WHERE** clause is a full legal operation in SQL.
Anyway SQL intends that the corresponding change must indiscriminately affect any row within the table: and sometimes this is exactly what you intended to do.

But (*much more often*) this is a wonderful way allowing to ***unintentionally*** destroy or to irreversibly corrupt your data: **beginner, pay careful attention**.

Recipe #8: Understanding Constraints

2011 January 28

Understanding what **constraints** are is a very simple task following a ***conceptual*** approach.
But on the other side understanding why some SQL statement will actually fail raising a generic ***constraint failed*** exception isn't a so simple affair.

In order to let you understand better this paragraph is structured like a **quiz**:

- you'll find first the **questions**.
- corresponding **answers** are positioned at bottom.

Important notice:
in order to preserve your *main* sample database untouched, creating a different database for this session is strongly suggested.

GETTING STARTED

```
CREATE TABLE mothers (
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  CONSTRAINT pk_mothers
    PRIMARY KEY (last_name, first_name));
```

```
SELECT AddGeometryColumn('mothers', 'home_location',
  4326, 'POINT', 'XY', 1);
```

```
CREATE TABLE children (
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  mom_first_nm TEXT NOT NULL,
  mom_last_nm TEXT NOT NULL,
  gender TEXT NOT NULL
    CONSTRAINT sex CHECK (
      gender IN ('M', 'F')),
  CONSTRAINT pk_childs
    PRIMARY KEY (last_name, first_name),
  CONSTRAINT fk_childs
    FOREIGN KEY (mom_last_nm, mom_first_nm)
      REFERENCES mothers (last_name, first_name));
```

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Stephanie', 'Smith',
    ST_GeomFromText('POINT(0.8 52.1)', 4326));
```

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Antoinette', 'Dupont',
    ST_GeomFromText('POINT(4.7 45.6)', 4326));
```

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Maria', 'Rossi',
    ST_GeomFromText('POINT(11.2 43.2)', 4326));
```

```
INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('George', 'Brown', 'Stephanie', 'Smith', 'M');

INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Janet', 'Brown', 'Stephanie', 'Smith', 'F');

INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Chantal', 'Petit', 'Antoinette', 'Dupont', 'F');

INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Henry', 'Petit', 'Antoinette', 'Dupont', 'M');

INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Luigi', 'Bianchi', 'Maria', 'Rossi', 'M');
```

Nothing too much complex: we simply have created two tables:

- the **mothers** table contains a Geometry column
- a relation exists between **mothers** and **children**: and consequently a **FOREIGN KEY** has been defined.
- just a last point to be noted: in this example we'll use **PRIMARY KEY**s spanning across two columns: but there isn't nothing odd in this ... it's a fully legitimate SQL option.

And then we have inserted very few rows into these tables.

```
SELECT m.last_name AS MomLastName,
  m.first_name AS MomFirstName,
  ST_X(m.home_location) AS HomeLongitude,
  ST_Y(m.home_location) AS HomeLatitude,
  c.last_name AS ChildLastName,
  c.first_name AS ChildFirstName,
  c.gender AS ChildGender
FROM mothers AS m
JOIN children AS c ON (
  m.first_name = c.mom_first_nm
  AND m.last_name = c.mom_last_nm);
```

MomLastName	MomFirstName	HomeLongitude	HomeLatitude	ChildLastName	ChildFirstName	ChildGender
Smith	Stephanie	0.8	52.1	Brown	George	M
Smith	Stephanie	0.8	52.1	Brown	Janet	F
Dupont	Antoinette	4.7	45.6	Petit	Chantal	F
Dupont	Antoinette	4.7	45.6	Petit	Henry	M
Rossi	Maria	11.2	43.2	Bianchi	Luigi	M

Just a simple check; and then you are now ready to start.

QUESTIONS

Q1: why this SQL statement will actually fail, raising a **constraint failed** exception ?

```
INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm)
  VALUES ('Silvia', 'Bianchi', 'Maria', 'Rossi');
```

Q2: *... same question ...*

```
INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Silvia', 'Bianchi', 'Maria', 'Rossi', 'f');
```

Q3: *... same question ...*

```
INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Silvia', 'Bianchi', 'Giovanna', 'Rossi', 'F');
```

Q4: ... *same question* ...

```
INSERT INTO children
    (first_name, last_name, mom_first_nm, mom_last_nm, gender)
  VALUES ('Henry', 'Petit', 'Stephanie', 'Smith', 'M');
```

Q5: ... *same question* ...

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Pilar', 'Fernandez',
    ST_GeomFromText('POINT(4.7 45.6)'));
```

Q6: ... *same question* ...

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Pilar', 'Fernandez',
    ST_GeomFromText('MULTIPOINT(4.7 45.6, 4.75 45.32)', 4326));
```

Q7: ... *same question* ...

```
INSERT INTO mothers (first_name, last_name)
  VALUES ('Pilar', 'Fernandez');
```

Q8: ... *same question* ...

```
INSERT INTO mothers (first_name, last_name, home_location)
  VALUES ('Pilar', 'Fernandez',
    ST_GeomFromText('POINT(4.7 45.6), 4326'));
```

Q9: ... *same question* ...

```
DELETE FROM mothers
WHERE last_name = 'Dupont';
```

Q10: ... *same question* ...

```
UPDATE mothers SET first_name = 'Marianne'
WHERE last_name = 'Dupont';
```

ANSWERS

A1: missing/undefined **`gender`**: so a **`NULL`** value is implicitly assumed.
But a **`NOT NULL`** constraint has been defined for the **`gender`** column.

A2: wrong **`gender`** value (**`'f'`**): SQLite text strings are ***case-sensitive***.
The **`sex`** constraint can only validate **`'M'`** or **`'F'`** values: **`'f'`** isn't an acceptable value.

A3: FOREIGN KEY failure.
No matching entry **`{'Rossi','Giovanna'}`** was found into the corresponding table [**`mothers`**].

A4: PRIMARY KEY failure.
An entry **`{'Petit','Henry'}`** is already stored into the **`children`** table.

A5: missing/undefined **`SRID`**: so a **`-1`** value is implicitly assumed.
But a **Geometry** constraint has been defined for the corresponding column; an explicitly set **`4326 SRID`** value is expected anyway for **`home_location`** geometries.

A6: wrong Geometry-type: only **`POINT`**-type Geometries will pass validation for the **`home_location`** column.

A7: missing/undefined **`home_location`**: so a **`NULL`** value is implicitly assumed.
But a **`NOT NULL`** constraint has been defined for the **`home_location`** column.

A8: malformed WKT expression: **`ST_GeomFromText()`** will return **`NULL`** (*same as above*).

A9: FOREIGN KEY failure: yes, the **`mothers`** table has no FOREIGN KEY.
But the **`children`** table instead has a corresponding FOREIGN KEY.
Deleting this entry from **`mothers`** will break ***referential integrity***, so this one isn't an allowed operation.

A10: FOREIGN KEY failure: more or less, the same of before.
Modifying a PRIMARY KEY entry into the **`mothers`** table will break ***referential integrity***, so this operation as well isn't admissible.

Lesson to learn #1:
An appropriate use of SQL constraints strongly helps to fully preserve your data in a well checked and absolutely consistent state.
Anyway, defining too much constraints may easily transform your database into a kind of inexpugnable fortress surrounded by trenches, pillboxes, barbed wire and minefields.
i.e. into something that surely nobody will define as **user friendly**.
Use sound common sense, and possibly avoid any excess.

Lesson to lean #2:
Each time the ***SQL-engine*** detects some constraint violation, a **constraint failed** exception will be immediately raised.
But this one is an absolutely **generic** error condition: so you have to use your experience and skilled knowledge in order to correctly understand (*and possibly resolve*) any possible glitch.

Recipe #9: ACIDity: undestranding Transactions

2011 January 28

ACID has nothing to do with chemistry (***pH***, *hydrogen and hydroxide ions and so on*).
In the DBMS context this one is an acronym meaning:

- **A***tomicity*
- **C***onsistency*
- **I***nsulation*
- **D***urability*

Very simply explained:

- a DBMS is designed to store complex data: sophisticated relations and constraints have to be carefully checked and validated.
  Data self-consistency has to be strongly preserved anyway.
- each time an **`INSERT`**, **`UPDATE`** or **`DELETE`** statement is performed, data self-consistency is at risk.
  If one single change fails (*for any reason*), this may leave the whole DB in an inconsistent state.
- any properly **`ACID`** compliant DBMS brilliantly resolves any such potential issue.

The underlying concept is based on a **`TRANSACTION`**-based approach:

- a **`TRANSACTION`** encloses an arbitrary group of SQL statements.
- a **`TRANSACTION`** is granted to be performed as an **`atomic`** unit, adopting an **all-or-nothing** approach.
  - if any statement enclosed within a **`TRANSACTION`** successfully completes, than the **`TRANSACTION`** itself can successfully complete.
  - but if a single statement fails, then the whole **`TRANSACTION`** will fail: and the DB will be left exactly in the previous state, as it was before the **`TRASACTION`** started.
- that's not all: any change occurred in a **`TRANSACTION`** context is absolutely invisible to any other DBMB connection, because a **`TRANSACTION`** defines an *insulated* private context.

Anyway, performing some direct test surely is the simplest way to understand **`TRANSACTION`**s.

```
BEGIN;

CREATE TABLE test (
  num INTEGER,
  string TEXT);

INSERT INTO test (num, string)
  VALUES (1, 'aaaa');

INSERT INTO test (num, string)
  VALUES (2, 'bbbb');
```

```
INSERT INTO test (num, string)
  VALUES (3, 'cccc');
```

The **BEGIN** statement will start a **TRANSACTION**:

- after this declaration, any subsequent statement will be handled within the current **TRANSACTION** context.
- you can also use the **BEGIN TRANSACTION** *alias*, but this is redundantly verbose, and not often used.
- SQLite forbids multiple *nested transactions*: you can simply declare an unique pending **TRANSACTION** at each time.

```
SELECT *
FROM test;
```

You can now check your work: there is nothing odd in this, isn't ?
Absolutely anything looks as expected.

Anyway, some relevant consequence arises from the initial **BEGIN** declaration:

- now you have a still pending (*unfinished*, *not completed*) **TRANSACTION**
- you can perform a first simple check:
  - open a second **spatialite_gui** instance, connecting the same DB
  - are you able to see the **test** table ?
  - NO: because this table has been created in the private (*insulated*) context of the first **spatialite_gui** instance, and so for any other different connection this table simply *does not yet exists*.
- and than you can perform a second check:
  - quit both **spatialite_gui** instances.
  - then launch again **spatialite_gui**.
  - there is no **test** table at all: it seems disappeared, completely vanishing.
  - but all this is easily explained: the corresponding **TRANSACTION** was never confirmed.
  - and when the holding connection terminated, then SQLite invalidated any operation within this **TRANSACTION**, so to leave the DB exactly in the previous state.

```
COMMIT;
```

```
ROLLBACK;
```

Once you **BEGIN** a **TRANSACTION**, any subsequent statement will be left in a *pending* (*uncommitted*) state.
Before or after you are expected to:

- close positively the **TRANSACTION** (*confirm*), by declaring a **COMMIT** statement.
  - any change applied to the DB will be confirmed and consolidated in a definitive way.
  - such changes will become immediately visible to other connections.
- close negatively the **TRANSACTION** (*invalidate*), by declaring a **ROLLBACK** statement.
  - any change applied to the DB will be rejected: the DB will be reverted to the previous state.
- if you omit declaring neither **COMMIT** nor **ROLLBACK**, then SQLite prudentially assumes that the still pending **TRANSACTION** is an invalid one, and a **ROLLBACK** will be implicitly performed.
- if any error or exception is encountered within a **TRANSACTION** context, than the whole **TRANSACTION** is invalidated, and a **ROLLBACK** is implicitly performed.

Performance Hints

Handling **TRANSACTION**s seems too much complex to you ? so you are thinking "*I'll simply ignore all this ...*"
Well, carefully consider that SQLite is a full **ACID DBMS**, so it's purposely designed to handle **TRANSACTION**s. And that's not all.

SQLite actually is completely unable to operate outside a **TRANSACTION** context.
Each time you miss to explicitly declare some **BEGIN / COMMIT**, then SQLite implicitly enters the so called **AUTOCOMMIT** mode:
each single statement will be handled as a self-standing **TRANSACTION**.

i.e. when you declare e.g. some simple **INSERT INTO ...** statement, then SQLite silently translates this into:

```
BEGIN;
INSERT INTO ...;
COMMIT;
```

Please note well: this is absolutely safe and acceptable when you are inserting few rows *by hand-writing*.
But when some C / C++ / Java / Python process attempts to **INSERT** many and many rows (maybe many ***million*** rows), this will impose an unacceptable overhead.
In other words, your process will perform very poorly, taking an unneeded long time to complete: and all this is simply caused by **not declaring** an explicit **TRANSACTION**.

The *strongly suggested* way to perform fast **INSERT**s (**UPDATE**, **DELETE** ...) is the following one:

- explicitly start a **TRANSACTION** (**BEGIN**)
- loop on **INSERT** as long as required.
- confirm the pending **TRANSACTION** (**COMMIT**).

And this simple trick will grant you very brilliant performances.

Connectors oddities (*true life tales*)

Developers, be warned: *different languages, different connectors, different default settings ...*

C / C++ developers will directly use the SQLite's API: in this environment the developer is expected to explicitly declare **TRANSACTION**s as required, by calling:

- `sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);`
- `sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);`

Java / JDBC connectors more or less follow the same approach: the developer is expected to explicitly quit the **AUTOCOMMIT** mode, then declaring a **COMMIT** when required and appropriate:

- `conn.setAutoCommit(false);`
- `conn.commit();`

Shortly said: in C / C++ and Java the developer is required to start a **TRANSACTION** in order to perform fast DB **INSERT**s.
Omitting this step will cause very slow performance. But at least any change will surely affect the underlying DB.

Python follows a completely different approach: a **TRANSACTION** is silently active at each time.
Performance always is optimal.
But forgetting to explicitly call **conn.commit()** before quitting, any applied change will be lost forever immediately after terminating the connection.
And this may really be puzzling for beginners, I suppose.

Recipe #10: Wonderful R*Tree Spatial Index

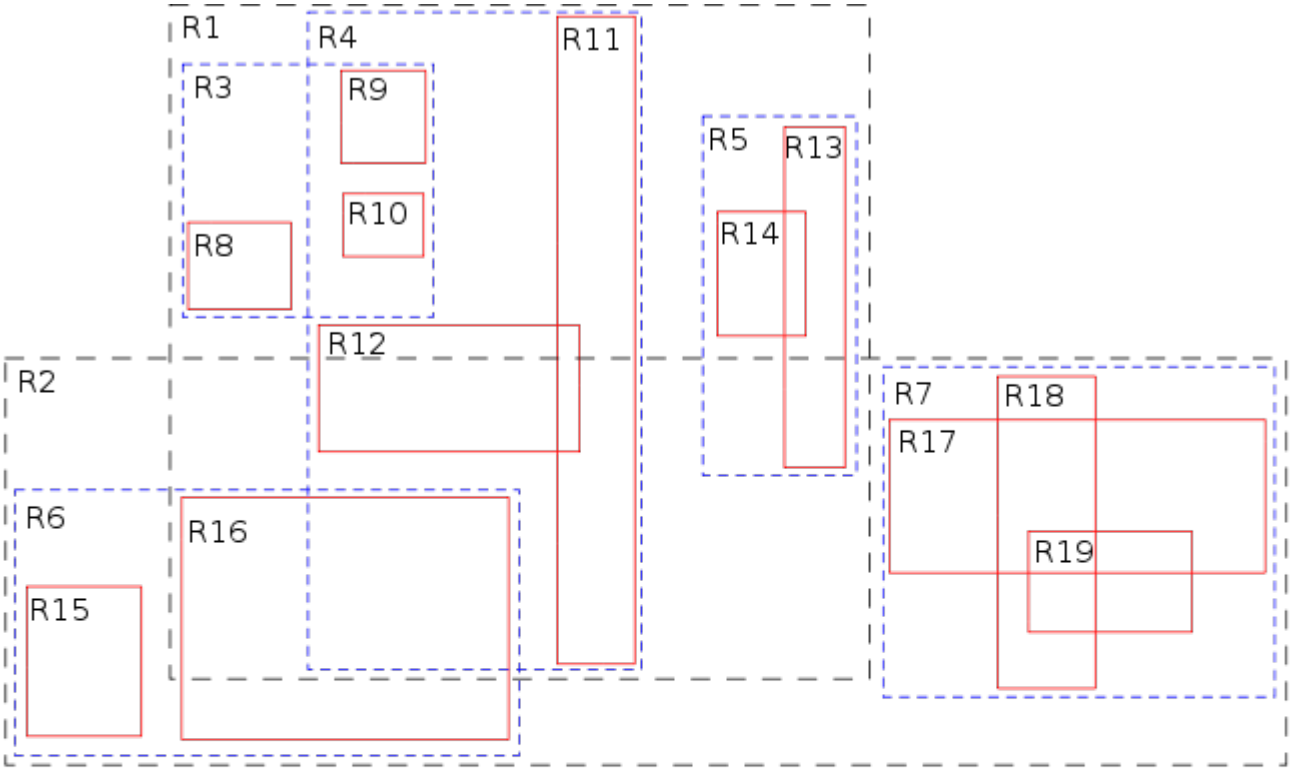
2011 January 28

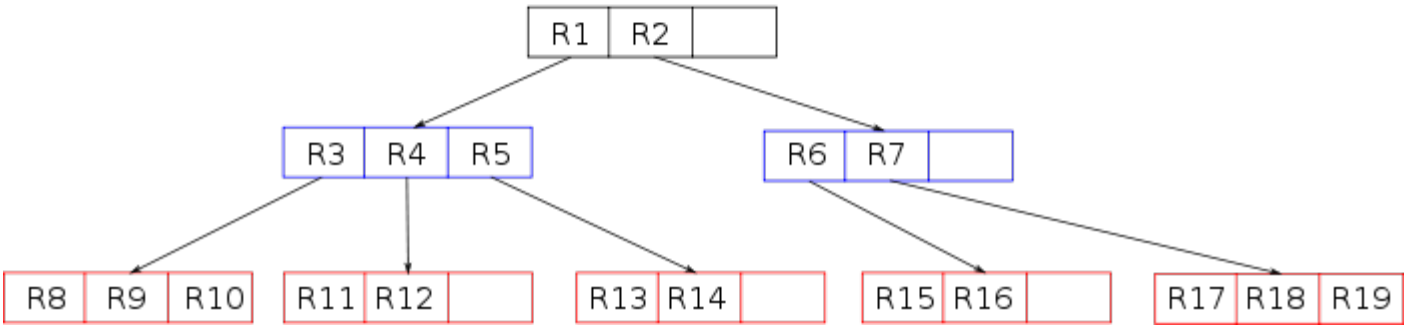
A **Spatial Index** more or less is like any other Index: i.e. the intended role of any Index is to support really fast search of selected items within an huge dataset.

Simply think of some huge textbook: searching some specific item by reading the whole book surely is painful, and may require a very long time.
But you can actually look at the textbook's index, then simply jumping to the appropriate page(s).

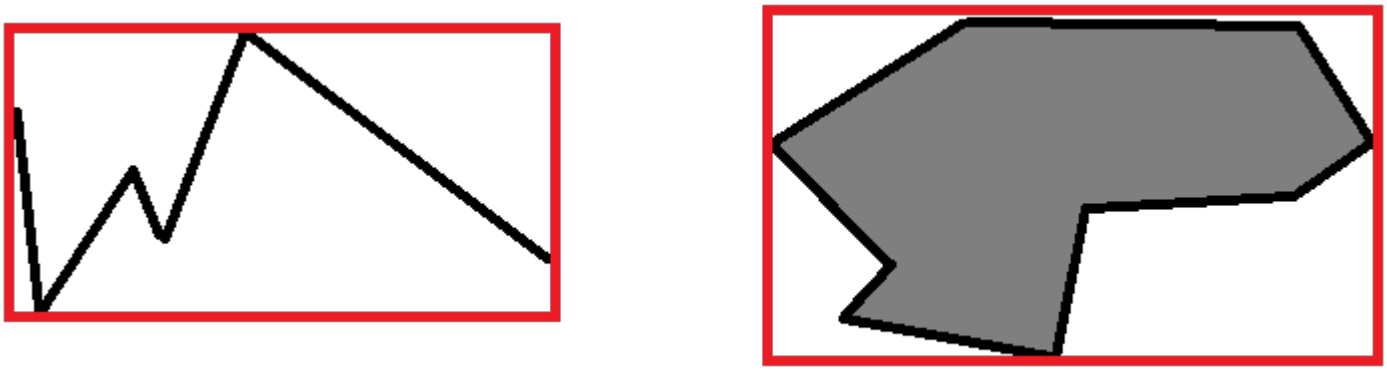
Any DB index plays exactly the same identical role.
Anyway, searching Geometries falling within a given search frame isn't the same of searching a text string or a number: so a different Index type is required. i.e. a **Spatial Index**.

Several algorithms supporting a Spatial Index has been defined during the past years.
SQLite's Spatial Index is based on the **R*Tree** algorithm.





Very shortly said, an R*Tree defines a *tree-like* structure based on rectangles (the **R** in R*Tree stands exactly for **R***ectangle*).



Every arbitrary Geometry can be represented as a *rectangle*, irrelevantly of its actual shape: we can simply use the **MBR** (*Minimum Bounding Rectangle*) corresponding to such Geometry.
May well be the term **BBOX** (*Bounding Box*) is more familiar to you: both terms are exact synonyms.

It's now quite intuitive understanding how the R*Tree does actually works:

- Spatial query defines an arbitrary search frame (this too being a rectangle)
- the R*Tree is quickly scanned identifying any overlapping index rectangle

- and finally any individual Geometry falling withing the search frame will be identified.>

Think of the well known ***needle in the hay*** problem: using an R*Tree is an excellent solution allowing to find the ***needle*** in a very short time, even when the ***hay*** actually is an impressively huge one.

Common misconceptions and misunderstandings

"*I have a table storing several zillion points disseminated all around the world:*
drawing a map was really painful and required a very long time.
Then I found somewhere some useful hint, so I've created a Spatial Index on this table.
And now my maps are drawn very quickly, as a general case.
Anyway I'm strongly puzzled, because drawing a worldwide map still takes a very long time.
Why the Spatial Index doesn't work on worldwide map ?"

The answer is elementary simple: the Spatial Index can speed up processing only when a small selected portion of the dataset has to be retrieved.
But when the whole (or a very large part of) dataset has to be retrieved, obviously the Spatial Index cannot give any speed benefit.
To be pedantic, under such conditions using the Spatial Index introduces further ***slowness***, because inquiring the R*Tree imposes a strong overhead.

Conclusion: the Spatial Index isn't a ***magic wand***. The Spatial Index basically is like a filter.

- when the selected frame covers a very small region of the whole dataset, using the Spatial Index implies a *ludicrous gain*.
- when the selected region covers a wide region, using the Spatial Index implies a *moderate gain*.
- but when the selected region covers the whole dataset (or nearly covers the whole dataset), using the Spatial Index implies a *further cost*.

SQLite's R*Tree implementation details

SQLite supports a first class R*Tree: anyway, some implementation details surely may seem strongly *exotic* for users accustomed to other different Spatial DBMS (such as PostGIS and so on).

Any R*Tree on SQLite actually requires **four** strictly correlated tables:

- ***rtreebasename*_node** stores (*binary format*) the R*Tree elementary nodes.
- ***rtreebasename*_parent** stores relations connecting parent and child nodes.
- ***rtreebasename*_rowid** stores ROWID values connecting an R*Tree node and a corresponding row into the indexed table.
  - none of these three tables is intended to be directly accessed: they are reserved for internal management.
- ***rtreebasename*** actually is a Virtual Table, and exposes the R*Tree for any external access.
  - **important notice:** never attempt to directly bungle or botch any R*Tree related table; quite surely such attempt will simply irreversibly corrupt the R*Tree. **You are warned**.

```
SELECT *
FROM rtreebasename;
```

pkuid	miny	maxx	miny	maxy
1022	313361.000000	331410.531250	4987924.000000	5003326.000000
1175	319169.218750	336074.093750	4983982.000000	4998057.500000
1232	329932.468750	337638.812500	4989399.000000	4997615.500000
...	...	...	...	...

Any R*Tree table looks like this one:

- The **pkid** column contains **ROWID** values.
- **minx**, **maxx**, **miny** and **maxy** defines **MBR** extreme points.

The R*Tree internal logic is *magically* implemented by the Virtual Table.

SpatiaLite's support for R*Tree

Any SpatiaLite Spatial Index fully relies on a corresponding SQLite R*Tree.
Anyway SpatiaLite smoothly integrates the R*Tree, so to make table handling absolutely painless:

- each time you perform an **INSERT**, **UPDATE** or **DELETE** affecting the *main table*, then SpatiaLite automatically take care to correctly reflect any change into the corresponding R*Tree.
- some ***triggers*** will grant such synchronization.
- so, once you've defined a Spatial Index, you can completely forget it.

Any SpatiaLite's Spatial Index always adopts the following naming convention:

- assuming a table named **local_councils** containing the **geometry** column.
- the corresponding Spatial Index will be named **idx_local_councils_geometry**
- and **idx.local_councils.pkid** will relationally reference **local_councils.ROWID**.

Anyway using the Spatial Index so to speed up Spatial queries execution is a little bit more difficult than in other Spatial DBMS, because there is no tight integration between the *main table* and the corresponding R*Tree: in the SQLite's own perspective they simply are two distinct tables.

Accordingly to all this, using a Spatial Index requires performing a **JOIN**, and (*may be*) defining a ***sub-query***.
You can find lots of examples about Spatial Index usage on SpatiaLite into the Haute Cuisine section.

```
SELECT CreateSpatialIndex('local_councils', 'geometry');

SELECT CreateSpatialIndex('populated_places', 'geometry');
```

This simple declaration is all you are required to specify in order to set a Spatial Index corresponding to some Geometry column. And that's all.

```
SELECT DiscardSpatialIndex('local_councils', 'geometry');
```

And this will remove a Spatial Index:

- **please note:** this will not **DROP** the Spatial Index (*you must perform this operation in a separate step*).
- anyway related *metadata* are set so to discard the Spatial Index, and any related **TRIGGER** will be immediately removed.

SpatiaLite supports a second alternitive Spatial Index based on ***MBR-caching***.
This one simply is a historical legacy, so using *MBR-caching* is ***strongly discouraged***.

recipe #11
Guinness Book of Records

2011 January 28

Local Council	County	Region
ATRANI	SALERNO	CAMPANIA
BARDONECCHIA	TORINO	PIEMONTE
BRIGA ALTA	CUNEO	PIEMONTE
CASAVATORE	NAPOLI	CAMPANIA
LAMPEDUSA E LINOSA	AGRIGENTO	SICILIA
LU	ALESSANDRIA	PIEMONTE
MORTERONE	LECCO	LOMBARDIA
NE	GENOVA	LIGURIA
OTRANTO	LECCE	PUGLIA
PINO SULLA SPONDA DEL LAGO MAGGIOR	VARESE	LOMBARDIA
PREDOI	BOLZANO	TRENTINO-ALTO ADIGE
RE	VERBANO-CUSIO-OSSOLA	PIEMONTE
RO	FERRARA	EMILIA-ROMAGNA
ROMA	ROMA	LAZIO
SAN VALENTINO IN ABRUZZO CITERIORE	PESCARA	ABRUZZO
VO	PADOVA	VENETO

The above list of Local Councils really is a kind of **Guinness Book of Records**.
For one reason or the other each one of them really has something absolutely exceptional and worth of note.
May well be you are really puzzled and surprised while reading this, because (*with the notable exception of Rome*) none of them is within the most renowned places of Italy.
Anyway, the explanation is really simple:

- **Roma** has the biggest population: **2,546,804** peoples
- **Morterone** has the smallest population: **33** peoples
- **Roma** (*again*) has the biggest area: **1,287** km²
- **Atrani** has the smallest area: **0.1** km²
- **Casavatore** has the highest population density: **13,627.5** peoples/km²
- **Briga Alta** has the lowest population density: **1.2** peoples/km²
- **Pino sulla sponda del Lago Maggior**(e) and **San Valentino in Abruzzo Citeriore** share the privilege to have the longest name.
- on the other side **Lu**, **Ne**, **Re**, **Ro** and **Vo** have the shortest name.
- **Predoi** is the northernmost
- **Lampedua e Linosa** is the southernmost
- **Bardonecchia** is the westernmost
- **Otranto** is the easternmost

Discovering such highly (*un*)useful Guinness Records collection is quite easy. You simply have to execute the following SQL query.

```
SELECT lc.lc_name AS LocalCouncil,
  c.county_name AS County,
  r.region_name AS Region,
  lc.population AS Population,
  ST_Area(lc.geometry) / 1000000.0 AS "Area sqKm",
  lc.population / (ST_Area(lc.geometry) / 1000000.0)
    AS "PopDensity [peoples/sqKm]",
  Length(lc.lc_name) AS NameLength,
  MbrMaxY(lc.geometry) AS North,
  MbrMinY(lc.geometry) AS South,
  MbrMinX(lc.geometry) AS West,
  MbrMaxX(lc.geometry) AS East
FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (
    SELECT lc_id
    FROM local_councils
    WHERE population IN (
      SELECT Max(population)
      FROM local_councils
    UNION
      SELECT Min(population)
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE ST_Area(geometry) IN (
      SELECT Max(ST_area(geometry))
      FROM local_councils
    UNION
      SELECT Min(ST_Area(geometry))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE population / (ST_Area(geometry) / 1000000.0) IN (
      SELECT Max(population / (ST_Area(geometry) / 1000000.0))
      FROM local_councils
    UNION
      SELECT MIN(population / (ST_Area(geometry) / 1000000.0))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE Length(lc_name) IN (
      SELECT Max(Length(lc_name))
      FROM local_councils
    UNION
      SELECT Min(Length(lc_name))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxY(geometry) IN (
      SELECT Max(MbrMaxY(geometry))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinY(geometry) IN (
      SELECT Min(MbrMinY(geometry))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxX(geometry) IN (
      SELECT Max(MbrMaxX(geometry))
      FROM local_councils)
  UNION
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinX(geometry) IN (
      SELECT Min(MbrMinX(geometry))
      FROM local_councils));
```

Oh yes, this one surely isn't a simple and plain query.
But you are now consulting the **High Cuisine Recipes**.
So I suppose your intention was exactly the one to look for some really tasty and spicy SQL query:
and you've just got it. Feel happy.

After all the above SQL query is only apparently complex, but it real structure is surprisingly simple.
Let try rewriting the SQL query in a simplified form:

```
SELECT lc.lc_name AS LocalCouncil,
  c.county_name AS County,
  r.region_name AS Region,
  lc.population AS Population,
  ST_Area(lc.geometry) / 1000000.0 AS "Area sqKm",
  lc.population / (ST_Area(lc.geometry) / 1000000.0)
    AS "PopDensity [peoples/sqKm]",
  Length(lc.lc_name) AS NameLength,
  MbrMaxY(lc.geometry) AS North,
  MbrMinY(lc.geometry) AS South,
  MbrMinX(lc.geometry) AS West,
  MbrMaxX(lc.geometry) AS East
FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (... some list of values ...);
```

I suppose you can now easily understand what is the meaning of this SQL query.
There is absolutely nothing too much complex or difficult in this:

- the **`local_councils`** table is **`JOIN`**ed to the **`counties`** table.
- and the **`counties`** table is then **`JOIN`**ed to the **`regions`** table.
- the **`ST_Area()`** Spatial function is used to calculate areas: an appropriate scale factor is applied so to get measures expressed in **km²** rather than in **m²**.
- a numeric expression (*not at all sophisticated or complex*) is used to calculate population density expressed in **peoples/km²**
- a **`WHERE ... IN (...)`** clause is then used so to filter someway the result-set.

But you already know all this, so I suppose you are very little interested to get any further detail.
Quite obviously the most interesting things happen within the **`WHERE ... IN (...)`** clause;
and exactly here we'll now focus our attention.

```
...
SELECT Max(population)
FROM local_councils
...
SELECT Min(population)
FROM local_councils
...
```

This SQL snippet is really simple: each query simply computes a **Min** / **Max** value.

```
...
  SELECT Max(population)
  FROM local_councils
UNION
  SELECT Min(population)
  FROM local_councils
...
```

This is the first time we introduce an **`UNION`** statement:

- this allows ***merging*** together two result-sets, so to get an unique result-set.
- there is a condition to be satisfied: both the ***left-sided*** and the ***right-sided*** **`SELECT`**s must return the same number of columns, and corresponding columns must have the same value-type.

```
...
SELECT lc_id
FROM local_councils
WHERE population IN (
    SELECT Max(population)
    FROM local_councils
  UNION
    SELECT Min(population)
    FROM local_councils)
...
```

SQL supports a wonderful mechanism: the one known as **sub-query**.
You can define an ***inner*** query (*which is executed first*), then using any returned value into the ***outer*** (*main*) query.
Now the previous SQL snippet doesn't looks too much mysterious:

- in this case the **sub-query** actually is an **`UNION`** query.
- the ***left-sided*** query will return the **`Max(population)`** value.
- the ***right-sided*** query will return the **`Min(population)`** value.
- then **`UNION`** will merge both values into an unique result-set:
  and this one is the result-set returned by the (***inner***) ***sub-query***.
- so the **`WHERE population IN (...)`** clause in the (***outer***) ***main*** query will simply receive two **`population`**-values to be checked.
- after all this, the (***outer***) ***main*** query will return a result-set which actually simply is a list of **`lc_id`**

```
...
  SELECT lc_id
  FROM local_councils
  WHERE population IN (
      SELECT Max(population)
      FROM local_councils
    UNION
      SELECT Min(population)
      FROM local_councils)
UNION
  SELECT lc_id
  FROM local_councils
  WHERE ST_Area(geometry) IN
      SELECT Max(ST_area(geometry))
      FROM local_councils
    UNION
      SELECT Min(ST_Area(geometry))
      FROM local_councils)
...
```

Nothing forbid us to nest more than one single **`UNION`** clauses: this one is a fully legitimate option.
Accordingly to this, the above SQL snippet has to be interpreted as follows:

- we've already explained the ***leftmost*** query in the previous paragraph:
  this will return a result-set containing a list of **`lc_id`**-values (**Max/Min** **`population`**-values).
- the ***rightmost*** query performs a similar task:
  this will return another result-set containing a further list of **`lc_id`**-values (**Max/Min** **`ST_Area(geometry)`**-values).
- so the ***second-level*** **`UNION`** clause will simply return a longer list of **`lc_id`**-values, i.e. the one resulting by merging all together any previous intermediate result.
- and so on ...

```
SELECT lc.lc_name AS LocalCouncil,
  c.county_name AS County,
  r.region_name AS Region,
  lc.population AS Population,
  ST_Area(lc.geometry) / 1000000.0 AS "Area sqKm",
  lc.population / (ST_Area(lc.geometry) / 1000000.0)
    AS "PopDensity [peoples/sqKm]",
  Length(lc.lc_name) AS NameLength,
  MbrMaxY(lc.geometry) AS North,
```

```
  MbrMinY(lc.geometry) AS South,
  MbrMinX(lc.geometry) AS West,
  MbrMaxX(lc.geometry) AS East
FROM local_councils AS lc
JOIN counties AS c ON (c.county_id = lc.county_id)
JOIN regions AS r ON (r.region_id = c.region_id)
WHERE lc.lc_id IN (
--
-- a list of lc.lc_id values will be returned
-- by this complex sub-query
--
    SELECT lc_id
    FROM local_councils
    WHERE population IN (
--
-- this further sub-query will return
-- Min/Max POPULATION
--
      SELECT Max(population)
      FROM local_councils
    UNION
      SELECT Min(population)
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE ST_Area(geometry) IN (
--
-- this further sub-query will return
-- Min/Max ST_AREA()
--
      SELECT Max(ST_area(geometry))
      FROM local_councils
    UNION
      SELECT Min(ST_Area(geometry))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE population / (ST_Area(geometry) / 1000000.0) IN (
--
-- this further sub-query will return
-- Min/Max POP-DENSITY
--
      SELECT Max(population / (ST_Area(geometry) / 1000000.0))
      FROM local_councils
    UNION
      SELECT MIN(population / (ST_Area(geometry) / 1000000.0))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE Length(lc_name) IN (
--
-- this further sub-query will return
-- Min/Max NAME-LENGTH
--
      SELECT Max(Length(lc_name))
      FROM local_councils
    UNION
      SELECT Min(Length(lc_name))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxY(geometry) IN (
--
-- this further sub-query will return
-- Max NORTH
--
      SELECT Max(MbrMaxY(geometry))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinY(geometry) IN (
--
-- this further sub-query will return
-- Max SOUTH
--
```

```
      SELECT Min(MbrMinY(geometry))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE MbrMaxX(geometry) IN (
--
-- this further sub-query will return
-- Max WEST
--
      SELECT Max(MbrMaxX(geometry))
      FROM local_councils)
  UNION -- merging into first-level sub-query
    SELECT lc_id
    FROM local_councils
    WHERE MbrMinX(geometry) IN (
--
-- this further sub-query will return
-- Max EAST
--
      SELECT Min(MbrMinX(geometry))
      FROM local_councils));
```

According to SQL syntax, using two consecutive *hyphens* (`--`) you can mark a **comment**.
i.e. any further text until the next line terminator is absolutely ignored by the SQL parser.
Placing appropriate ***comments*** within really complex SQL queries surely enhances readibility.

Conclusion: SQL is a wonderful language, fully supporting a regular and easily predictable syntax.
Each time you'll encounter some intimidating complex SQL query don't panic and don't be afraid:
simply attempt to break the complex statement into several smallest and simplest blocks, and you'll soon discover that complexity was more apparent than real.

Useful hint: attempting to debug some very complex SQL statement is obviously a difficult and defatigating task.
Breaking down a complex query into smallest chunks, then testing each one of them individually usually is the best approach you can follow.

Recipe #12
Neighbours

2011 January 28

The problem

- Obviously each Local Council shares a common boundary with a neighbour:
  [*not an absolute rule, anyway: e.g. small islands are self-contained*].
- We'll use Spatial SQL to identify every adjacent couple of Local Councils.
- Just to add some further complexity, we'll specifically focus our attention on the Tuscany region boundaries.

Tuscan Local Council	Tuscan County	Neighbour LC	County	Region
ANGHIARI	AREZZO	CITERNA	PERUGIA	UMBRIA
AREZZO	AREZZO	MONTE SANTA MARIA TIBERINA	PERUGIA	UMBRIA
BIBBIENA	AREZZO	BAGNO DI ROMAGNA	FORLI' - CESENA	EMILIA-ROMAGNA
CHIUSI DELLA VERNA	AREZZO	BAGNO DI ROMAGNA	FORLI' - CESENA	EMILIA-ROMAGNA
CHIUSI DELLA VERNA	AREZZO	VERGHERETO	FORLI' - CESENA	EMILIA-ROMAGNA
...	...	...	...	...

```
SELECT lc1.lc_name AS "Local Council",
  lc2.lc_name AS "Neighbour"
FROM local_councils AS lc1,
  local_councils AS lc2
WHERE ST_Touches(lc1.geometry, lc2.geometry);
```

This first query is really simple:

- the **`ST_Touches(`*`geom1, geom2`*`)`** function is used to evaluates the spatial relationship existing between couples of Local Councils.
- the **`local_council`** table is scanned twice, so to implement a **`JOIN`**;
  in other words this simply allows to evaluate each Local Council against any other, in a combinatory/permutative fashion.
- obviously this may cause ambiguity.
  We have to set an appropriate *alias* name (**`AS lc1`** / **`AS lc2`**) so to uniquely identify each one of the two table instances.

Anyway, a so simplistic approach implies several (*strong, severe*) issues:

- this query surely will return the correct answer: but the process time will be very long.
  [*actually, so long to be absolutely not usable for any practical purpose*].
- explaining all this is really easy: evaluating **`ST_Touches()`** implies lots of complex calculations, so this one is a really heavy (*and lengthy*) step.
- following a pure combinatorial logic creates many million couples requiring to be evaluated.
- conclusion: iterating many million times a lengthy operation is a sure recipe for disaster.

```
SELECT lc1.lc_name AS "Local Council",
  lc2.lc_name AS "Neighour"
FROM local_councils AS lc1,
  local_councils AS lc2
WHERE lc2.ROWID IN (
  SELECT pkid
    FROM idx_local_councils_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(lc1.geometry),
      MbrMinY(lc1.geometry),
      MbrMaxX(lc1.geometry),
      MbrMaxY(lc1.geometry)));
```

Happily enough, we can perform such Spatial queries in a really fast and efficient way:

- we have to use a **Spatial Index** [aka ***R*Tree***]
- this will add some further complexity to the SQL statement, but will achieve a ludicrous speed enhancement.
- how it works: the ***R*Tree*** is checked first, so to evaluate *Minimum Bounding Rectangles* [**MBR**s] of both Geometries.
  - This one is a very quick step to be evaluated, and allows discarding lots of couples that surely cannot share a common boundary.
  - So the number of times **`ST_Touches()`** will then be actually called will be dramatically reduced.
  - And all this strongly reduces the overall processing time.
- At SQL syntax level using the Spatial Index simply requires to implement a **sub-query**:
  - the ***R*Tree Spatial Index*** in SQLite actually is a separate table.
  - table names are strictly related: the Spatial Index corresponding to table **`myTbl`** and geometry column **`myGeom`** always is expected to be named as **`idx_myTbl_myGeom`**.
  - the **`MATCH RTreeIntersects()`** clause is used to quickly retrieve any interesting Geometry simply evaluating its own MBR.
  - **MbrMinX()** and alike are used to identify the extreme points of the *filtering* MBR.

Just to explain better what's going on, you can imagine that this SQL query is processed using the following steps:

- a Geometry is picked up from the first **`local_councils`** instance: **`lc1.geometry`**
- then the *R*Tree Spatial Index* is scanned, so to identify any other Geometry from the second **`local_councils`** instance: **`lc2.geometry`**
- Only Geometries satisfying an **intersecting MBR** constraint will be fetched *via Spatial Index*.
- and finally **`ST_Touches()`** will be evaluated: but this will affect only a very limited few carefully pre-filtered Geometries.

```
SELECT lc1.lc_name AS "Tuscan Local Council",
  c1.county_name AS "Tuscan County",
  lc2.lc_name AS "Neighbour LC",
  c2.county_name AS County,
  r2.region_name AS Region
FROM local_councils AS lc1,
  local_councils AS lc2,
  counties AS c1,
  counties AS c2,
  regions AS r1,
  regions AS r2
WHERE c1.county_id = lc1.county_id
  AND c2.county_id = lc2.county_id
  AND r1.region_id = c1.region_id
  AND r2.region_id = c2.region_id
  AND r1.region_name LIKE 'toscana'
  AND r1.region_id <> r2.region_id
  AND ST_Touches(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT pkid
      FROM idx_local_councils_geometry
      WHERE pkid MATCH RTreeIntersects(
        MbrMinX(lc1.geometry),
        MbrMinY(lc1.geometry),
        MbrMaxX(lc1.geometry),
```

```
        MbrMaxY(lc1.geometry)))
ORDER BY c1.county_name, lc1.lc_name;
```

All right, once we have resolved the Spatial Index stuff writing the whole SQL query isn't so difficult.
Anyway this one is a rather complex query, so some further explanation is surely welcome:

- we have to resolve **JOIN** relations connecting **`local_councils`** to **`counties`**, and **`counties`** to **`regions`**
- anyway we used two different instances for **`local_councils`**, so we need to resolve **JOIN** relations separately for each one instance.
- setting the **`r1.region_name LIKE 'toscana'`** clause only Tuscan Local Councils will be evaluated for the *homeland* side.
- and setting the **`r1.region_id <> r2.region_id`** clause ensures that only not-Tuscan Local Councils will be evaluated for the *foreigner* side.
- only Tuscan Local Councils sharing a common boundary with not-Tuscan Local Councils will be extracted by this query.

```
SELECT lc1.lc_name AS "Tuscan Local Council",
  c1.county_name AS "Tuscan County",
  lc2.lc_name AS "Neighbour LC",
  c2.county_name AS County,
  r2.region_name AS Region
FROM local_councils AS lc1,
  local_councils AS lc2
JOIN counties AS c1
  ON (c1.county_id = lc1.county_id)
JOIN counties AS c2
  ON (c2.county_id = lc2.county_id)
JOIN regions AS r1
  ON (r1.region_id = c1.region_id)
JOIN regions AS r2
  ON (r2.region_id = c2.region_id)
WHERE r1.region_name LIKE 'toscana'
  AND r1.region_id <> r2.region_id
  AND ST_Touches(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT pkid
      FROM idx_local_councils_geometry
      WHERE pkid MATCH RTreeIntersects(
        MbrMinX(lc1.geometry),
        MbrMinY(lc1.geometry),
        MbrMaxX(lc1.geometry),
        MbrMaxY(lc1.geometry)))
ORDER BY c1.county_name, lc1.lc_name;
```

Obviously you can this query adopting the alternative syntax for **JOIN**s: the difference simply is syntactic.
And doesn't implies any difference at functional or performance levels.

Performing sophisticated Spatial Analysis not necessarily is an easy and plain task.
Mastering complex SQL queries is a little bit difficult (*but not at all impossible*).
Optimizing such complex SQL, so to get fast answers surely requires some extra-care and attention.

But Spatial SQL supports you in the most effective (*and flexible*) way: the results you can get simply are fantastic.
After all **the game surely is worth the candle**.

Recipe #13
Isolated Islands

2011 January 28

The problem

Very closely related to the latest one. Now the problem is:

- identify any isolated Local Council, i.e. each one doesn't sharing a common boundary with any other Italian Local Councils.

Local Council	County	Region
CAMPIONE D'ITALIA	COMO	LOMBARDIA
CAPRAIA ISOLA	LIVORNO	TOSCANA
CARLOFORTE	CAGLIARI	SARDEGNA
FAVIGNANA	TRAPANI	SICILIA
ISOLA DEL GIGLIO	GROSSETO	TOSCANA
ISOLE TREMITI	FOGGIA	PUGLIA
LA MADDALENA	SASSARI	SARDEGNA
LAMPEDUSA E LINOSA	AGRIGENTO	SICILIA
LIPARI	MESSINA	SICILIA
PANTELLERIA	TRAPANI	SICILIA
PONZA	LATINA	LAZIO
PROCIDA	NAPOLI	CAMPANIA
USTICA	PALERMO	SICILIA
VENTOTENE	LATINA	LAZIO

Please note: quite all the above listed Local Councils are small sea island.
With the remarkable exception of **Campione d'Italia**, which is a ***land island***:
i.e. a small Italian ***enclave*** completely surrounded by Switzerland.

```
SELECT lc1.lc_name AS "Local Council",
  c.county_name AS County,
  r.region_name AS Region
FROM local_councils AS lc1
JOIN counties AS c ON (
  c.county_id = lc1.county_id)
JOIN regions AS r ON (
  r.region_id = c.region_id)
LEFT JOIN local_councils AS lc2 ON (
  lc1.lc_id <> lc2.lc_id
  AND NOT ST_Disjoint(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
```

```
    SELECT pkid
    FROM idx_local_councils_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(lc1.geometry),
      MbrMinY(lc1.geometry),
      MbrMaxX(lc1.geometry),
      MbrMaxY(lc1.geometry))))
GROUP BY lc1.lc_id
HAVING Count(lc2.lc_id) = 0
ORDER BY lc1.lc_name;
```

Nothing really new in this: more or less, this is quite exactly the same we've already examined in the latest example.
Just few differences are worth to be explained:

- this time we've used **`NOT ST_Disjoint()`** to identify any allowable Spatial relationship between couples of adjacent Local Councils.
- and we've used **`LEFT JOIN`** for the second instance of the **`local_councils`** table (**`AS lc2`**): this way we'll be absolutely sure to insert into the result-set every Local Council coming from the ***left-sided*** term **`lc1`**, because a **`LEFT JOIN`** is valid even when the ***right-sided*** term **`lc2`** doesn't matches any corresponding entry.
- the **`GROUP BY lc1.lc_id`** clause is required so to build a distinct aggregation group for each Local Council.
- after all this the function **`Count(lc2.lc_id)`** will return the number of neighbours for each Local Council: quite obviously, a value **`ZERO`** denotes that this one actually is an isolated Local Council.
- and finally we've used the **`HAVING`** clause to exclude any not-isolated Local Council.
- **Please note well:** the **`HAVING`** clause must not be confused with the **`WHERE`** clause.
  They only are apparently similar, but a strong difference exists between them:
  - **`WHERE`** immediately evaluates if a *candidate row* has to inserted into the result-set or not.
    So, a row discarded by **`WHERE`** is completely ignored, and cannot be used in any further step.
  - on the other side **`HAVING`** is evaluated only when the result-set is completely defined, just immediately before be passed back to the calling process.
    So **`HAVING`** is really useful to perform any kind of ***post-processing***, as in this case.
    We simply needed to ***reduce*** the result-set (*by deleting any not-isolated Local Council*), and the **`HAVING`** clause was exactly the tool for the job.

Recipe #14
Populated Places vs Local Councils

2011 January 28

The problem

Do you remember ?

- We've left the **`populated_places`** table in a self-standing position since now.
- While designing the DB layout we concluded that some spatial relationship must exists between **`populated_places`** and **`local_councils`**.
- We can easily expect to get some inconsistencies between these two datasets, because they come from absolutely unrelated sources.
- The **`populated_places`** table has **`POINT`** Geometries into the **`4236 SRID`** (**Geographic**, ***WGS84***, *long-lat*): whilst the **`local_councils`** table has **`MULTIPOLYGON`** Geometries into the **`23032 SRID`** (*planar*, ***ED50 UTM zone 32***)
- Using two different **SRID**s surely introduces some further complication to be resolved.

It's now time to confront yourself with this not-so-simple problem.

PopulatedPlaceId	PopulatedPlaceName	LocalCouncilId	LocalCouncilName	County	Region
...	...	...	...	...	...
12383	Acitrezza	NULL	NULL	NULL	NULL
12384	Lavinio	NULL	NULL	NULL	NULL
11327	Altino	69001	ALTINO	CHIETI	ABRUZZO
11265	Archi	69002	ARCHI	CHIETI	ABRUZZO
11247	Ari	69003	ARI	CHIETI	ABRUZZO
...	...	...	...	...	...

```
SELECT pp.id AS PopulatedPlaceId,
  pp.name AS PopulatedPlaceName,
  lc.lc_id AS LocalCouncilId,
  lc.lc_name AS LocalCouncilName
FROM populated_places AS pp,
  local_councils AS lc
WHERE ST_Contains(lc.geometry,
  Transform(pp.geometry, 23032));
```

You can start with this first simple query:

- the **`ST_Contains(`*`geom1, geom2`*`)`** function is used to evaluate the spatial relationship existing between Local Councils and Populated Places.
- there is absolutely nothing strange in this query: you'll simply use a JOIN condition based on a spatial

relationship.
- quite obviously using **`Transform()`** is required so to re-project any coordinate into the same SRID.
- anyway you are already warned: you are now well conscious that using a so simplistic approach (i.e. not using the *R*Tree Spatial Index*) will surely produce a very slowly running query.

```
SELECT pp.id AS PopulatedPlaceId,
  pp.name AS PopulatedPlaceName,
  lc.lc_id AS LocalCouncilId,
  lc.lc_name AS LocalCouncilName
FROM populated_places AS pp,
  local_councils AS lc
WHERE ST_Contains(lc.geometry,
    Transform(pp.geometry, 23032))
  AND lc.lc_id IN (
    SELECT pkid
    FROM idx_local_councils_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(pp.geometry, 23032)),
      MbrMinY(
        Transform(pp.geometry, 23032)),
      MbrMaxX(
        Transform(pp.geometry, 23032)),
      MbrMaxY(
        Transform(pp.geometry, 23032)));
```

This further query is exactly the same as the first one: except in that this second version fully exploits the *R*Tree Spatial Index*.
Please note: using **`Transform()`** several times is absolutely required, so to correctly re-project any coordinate into an uniform SRID.

Anyway there is still an unresolved issue in the above query: following this way any mismatching Populated Place will never be identified.
In order to detect if some Populated Place does actually falls outside any corresponding Local Council you absolutely have to implement a **`LEFT JOIN`**.

```
SELECT pp.id AS PopulatedPlaceId,
  pp.name AS PopulatedPlaceName,
  lc.lc_id AS LocalCouncilId,
  lc.lc_name AS LocalCouncilName,
  c.county_name AS County,
  r.region_name AS Region
FROM populated_places AS pp
LEFT JOIN local_councils AS lc
  ON (ST_Contains(lc.geometry,
      Transform(pp.geometry, 23032))
    AND lc.lc_id IN (
      SELECT pkid
      FROM idx_local_councils_geometry
      WHERE pkid MATCH RTreeIntersects(
        MbrMinX(
          Transform(pp.geometry, 23032)),
        MbrMinY(
          Transform(pp.geometry, 23032)),
        MbrMaxX(
          Transform(pp.geometry, 23032)),
        MbrMaxY(
          Transform(pp.geometry, 23032)))))
LEFT JOIN counties AS c
  ON (c.county_id = lc.county_id)
LEFT JOIN regions AS r
  ON (r.region_id = c.region_id)
ORDER BY 6, 5, 4;
```

All right: this one is the final and definitive version.

- you've simply added further LEFT JOIN clauses, so to fully qualify each Local Council reporting the corresponding County and Region.
- not at all surprisingly about twenty Populated Places doesn't actually correspond to any Local Council

(*you expected this, because these two datasets come from unrelated sources*).

- you can duly use QGIS to visually inspect such mismatching entries: and you'll soon discover that in each case all them are towns placed very closely to sea shore.
  And some (*slight*) misplacement actually exist.

recipe #15
Tightly bounded Populated Places

2011 January 28

The problem

Yet another problem based on the **`populated_places`** dataset. This time the question is:

- Identify any possible couple of Populated Places laying at very close distance: **< 1 Km**

Please note: this problem hides an unpleasant complication.

- the Populated Places dataset is in the **`4236 SRID`** (**Geographic**, ***WGS84***, *long-lat*)
- accordingly to this, distances are naturally measured in **decimal degrees**
- but the imposed range limit is expressed in **meters/Km**

PopulatedPlace #1	Distance (meters)	PopulatedPlace #2
Vallarsa	49.444299	Raossi
Raossi	49.444299	Vallarsa
Seveso	220.780551	Meda
Meda	220.780551	Seveso
...	...	...

```
SELECT pp1.name AS "PopulatedPlace #1",
  GeodesicLength(
    MakeLine(pp1.geometry, pp2.geometry))
      AS "Distance (meters)",
  pp2.name AS "PopulatedPlace #2"
FROM populated_places AS pp1,
  populated_places AS pp2
WHERE GeodesicLength(
    MakeLine(pp1.geometry, pp2.geometry)) < 1000.0
  AND pp1.id <> pp2.id
  AND pp2.ROWID IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeDistWithin(
      ST_X(pp1.geometry),
      ST_Y(pp1.geometry), 0.02))
ORDER BY 2;
```

This time we'll go straight forward to final solution.
I suppose that's now very clear to everyone that using a Spatial Index is absolutely required to get a decently well-performing query.
And that a JOIN between two different instances of the same table is required to perform this kind of Spatial

Analysis, and so on ...

So we'll simply focus our attention on the most notable highlights:

- the **`MakeLine()`** function builds a segment connecting two extreme **`POINT`**
- the **`GeodesicLength()`** function calculates the total length (expressed in **meters**) for any *long-lat* **`LINESTRING`**.
  This function gives very accurate results, because is taken directly on the **ellipsoid**.
  Unhappily, this requires lots of complex calculations, so computing a Geodesic length is intrinsically an heavy (*and slow*) process.
  Anyway, a savvy usage of the Spatial Index strongly reduces complexity.
- using the **`MATCH RTreeDistWithin()`** clause allows to make a first distance estimation directly on the Spatial Index.
  Please note: the **0.02** constant means **2/100 of degree** (*about 2Km, on the Great Circle*).
  Coordinates into the Spatial Index are ***long-lat*** (*this is because Populated Places are into the **4326 SRID***), so an angular measure is required here.
- defining the **`pp1.id <> pp2.id`** clause avoids to evaluate the distance between a Populated Place and itself (*not surprisingly, always exactly equal to **0.0***)

Performing a Spatial query like this one in the most *naive* way requires an extremely long time, even if you'll use the most recent and powerful CPU. But carefully applying a little bit of optimization is not too much difficult. And a properly defined an well optimized SQL query surely runs in the smoothest and fastest way.

Recipe #16
Railways vs Local Councils

2011 January 28

The problem

This time we'll use for the first time the **`railways`** dataset.
Please remember: this one is a really small dataset simply representing two railway lines:
this dataset is in the **`23032 SRID`** [**ED50 UTM zone 32**]. The problem is:

- Identify any Local Council crossed by a railway line.

Important notice: you must accomplish a preliminary step.
You are required downloading railways.zip (a very simple *shapefile* oppurtunely derived from OSM).
And then you have to load such *shapefile* into the **`railways`** table.

Railway	LocalCouncil	County	Region
...	...	...	...
Ferrovia Adriatica	SILVI	TERAMO	ABRUZZO
Ferrovia Adriatica	TORTORETO	TERAMO	ABRUZZO
Ferrovia Roma-Napoli	AVERSA	CASERTA	CAMPANIA
Ferrovia Roma-Napoli	CANCELLO ED ARNONE	CASERTA	CAMPANIA
...	...	...	...

```
SELECT rw.name AS Railway,
  lc.lc_name AS LocalCouncil,
  c.county_name AS County,
  r.region_name AS Region
FROM railways AS rw
JOIN local_councils AS lc ON (
  ST_Intersects(rw.geometry, lc.geometry)
    AND lc.ROWID IN (
      SELECT pkid
      FROM idx_local_councils_geometry
      WHERE pkid MATCH RTreeIntersects(
        MbrMinX(rw.geometry),
        MbrMinY(rw.geometry),
        MbrMaxX(rw.geometry),
        MbrMaxY(rw.geometry))))
JOIN counties AS c
  ON (c.county_id = lc.county_id)
JOIN regions AS r
  ON (r.region_id = c.region_id)
ORDER BY r.region_name,
  c.county_name,
  lc.lc_name;
```

We'll simply examine few interesting key points:

- a **`JOIN`** clause is used so to retrieve the corresponding County and Region for each Local Council.
  You already know how this works, because you had already used this in some previous example.
- the most interesting point in this query is in the first **`JOIN`** clause:
  the **`ST_Intersects()`** function is used to evaluate a Spatial relationships between the **`local_councils`** and the **`railways`** tables.
  Anyway all this isn't at all surprising, because you've already seen something like this in previous examples.
- and once again the appropriate ***R*Tree Spatial Index*** is used in order to speed up the query.

More or less, this is quite the same thing of the previous example, when we examined Spatial relationships existing between Local Councils and Populated Places.
Anyway, this confirms that using **any** possible kind of Spatial relationship is a reasonably easy task, and that you can successfully use Spatial relationships to resolve lots of different *real-world* problems.

Recipe #17
Railways vs Populated Places

2011 January 28

The problem

We'll use once again the **`railways`** dataset.
But this really is an ***hot spiced*** recipe: be prepared to taste ***very*** strong flavors.
As you can now easily image by yourself, computing distances between a railway line and Populated Places isn't so difficult.
So this problem introduces a further degree of complexity (*just to escape from boredom and to keep your mind active and interested*).

Image that for any good reason the following classification exists:

Class	Min. distance	Max. distance
A-class	0 Km	1 Km
B-class	1 Km	2.5 Km
C-class	2.5 Km	5 Km
D-class	5 Km	10 Km
E-class	10 Km	20 Km

The problem you are faced to resolve is:

- identify any Populated Place laying within a distance radius of 20 Km from a Railway.
- identify the corresponding distance Class for each one of such Populated Places.

Railway	PopulatedPlace	A class [< 1Km]	B class [< 2.5Km]	C class [< 5Km]	D class [< 10Km]	E class [< 20Km]
Ferrovia Adriatica	Zapponeta	NULL	NULL	NULL	NULL	1
Ferrovia Adriatica	Villamagna	NULL	NULL	NULL	NULL	1
Ferrovia Adriatica	Villalfonsina	NULL	NULL	NULL	1	0
Ferrovia Adriatica	Vasto	1	0	0	0	0
...	...	...	...	...	...	...

```
SELECT rw.name AS Railway,
  pp_e.name AS PopulatedPlace,
  (ST_Distance(rw.geometry,
    Transform(pp_a.geometry, 23032)) <= 1000.0)
      AS "A class [< 1Km]",
```

```
  (ST_Distance(rw.geometry,
    Transform(pp_b.geometry, 23032)) > 1000.0)
      AS "B class [< 2.5Km]",
  (ST_Distance(rw.geometry,
    Transform(pp_c.geometry, 23032)) > 2500.0)
      AS "C class [< 5Km]",
  (ST_Distance(rw.geometry,
    Transform(pp_d.geometry, 23032)) > 5000.0)
      AS "D class [< 10Km]",
  (ST_Distance(rw.geometry,
    Transform(pp_e.geometry, 23032)) > 10000.0)
      AS "E class [< 20Km]"
FROM railways AS rw
JOIN populated_places AS pp_e ON (
  ST_Distance(rw.geometry,
    Transform(pp_e.geometry, 23032)) <= 20000.0
  AND pp_e.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMinY(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMaxX(
        Transform(
          ST_Envelope(rw.geometry), 4236)),
      MbrMaxY(
        Transform(
          ST_Envelope(rw.geometry), 4326)))))
LEFT JOIN populated_places AS pp_d ON (
  pp_e.id = pp_d.id
  AND ST_Distance(rw.geometry,
    Transform(pp_d.geometry, 23032)) <= 10000.0
  AND pp_d.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMinY(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMaxX(
        Transform(
          ST_Envelope(rw.geometry), 4236)),
      MbrMaxY(
        Transform(
          ST_Envelope(rw.geometry), 4326)))))
LEFT JOIN populated_places AS pp_c ON (
  pp_d.id = pp_c.id
  AND ST_Distance(rw.geometry,
    Transform(pp_c.geometry, 23032)) <= 5000.0
  AND pp_c.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMinY(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMaxX(
        Transform(
          ST_Envelope(rw.geometry), 4236)),
      MbrMaxY(
        Transform(
          ST_Envelope(rw.geometry), 4326)))))
LEFT JOIN populated_places AS pp_b ON (
  pp_c.id = pp_b.id
  AND ST_Distance(rw.geometry,
    Transform(pp_b.geometry, 23032)) <= 2500.0
  AND pp_b.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
```

```
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMinY(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMaxX(
        Transform(
          ST_Envelope(rw.geometry), 4236)),
      MbrMaxY(
        Transform(
          ST_Envelope(rw.geometry), 4326)))))
LEFT JOIN populated_places AS pp_a ON (
  pp_b.id = pp_a.id
  AND ST_Distance(rw.geometry,
    Transform(pp_a.geometry, 23032)) <= 1000.0
  AND pp_a.id IN (
    SELECT pkid
    FROM idx_populated_places_geometry
    WHERE pkid MATCH RTreeIntersects(
      MbrMinX(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMinY(
        Transform(
          ST_Envelope(rw.geometry), 4326)),
      MbrMaxX(
        Transform(
          ST_Envelope(rw.geometry), 4236)),
      MbrMaxY(
        Transform(
          ST_Envelope(rw.geometry), 4326)))));
```

Yes, this one really looks like a complex and intimidating query.
Anyway, complexity is much more apparent than real.
You already know the trick: you simply have to break down this statement into several smallest chunks. And then you'll soon discover that there isn't nothing really difficult and complex.

Let us examine the main framework:

```
SELECT rw.name AS Railway, ...
FROM railways AS rw
JOIN populated_places AS pp_e ON (...)
LEFT JOIN populated_places AS pp_d ON (...)
LEFT JOIN populated_places AS pp_c ON (...)
LEFT JOIN populated_places AS pp_b ON (...)
LEFT JOIN populated_places AS pp_a ON (...);
```

- we'll simply **`JOIN`** the **`railways AS rw`** and the **`populated_places AS pp_e`** tables: and there is nothing strange in this, isn't ?
- then we'll **`LEFT JOIN`** the **`populated_places AS pp_d`** table a second time: and you surely remember that **`LEFT JOIN`** inserts a valid row into the result-set even when the ***right-sided*** term evaluates to **`NULL`**.
- and finally we'll repeat **LEFT JOIN** for **`pp_c`**, **`pp_b`** and **`pp_a`**.
- we'll obviously start by checking the biggest distance, because if this one fails any other comparison will surely fail as well.
  And so on, following the ***decreasing*** distance sequence.

```
...
JOIN populated_places AS pp_e ON (
  ST_Distance(rw.geometry,
    Transform(pp_e.geometry, 23032)) <= 20000.0
...
```

- each **`JOIN`** (or **`LEFT JOIN`**) simply evaluates the distance intercurring between the Railway line and the Populated Place, checking if this one falls within the expected threshold for the corresponding Class.
- using **`Transform()`** is required because the railway is in the **23032** SRID, whilst the populated place is in the **4326** SRID.

```
... AND pp_e.id IN (
  SELECT pkid
```

```
  FROM idx_populated_places_geometry
  WHERE pkid MATCH RTreeIntersects(
    MbrMinX(
      Transform(
        ST_Envelope(rw.geometry), 4326)),
    MbrMinY(
      Transform(
        ST_Envelope(rw.geometry), 4326)),
    MbrMaxX(
      Transform(
        ST_Envelope(rw.geometry), 4236)),
    MbrMaxY(
      Transform(
        ST_Envelope(rw.geometry), 4326)))
...
```

- using the usual Spatial Index handling stuff is required anyway, so to support fast filtering of Populated Places geometries.
- and obviously we have to apply **`Transform()`**, so to reproject the railway geometry into the **4326** SRID (*the one used by Populated Places*).
- **Please note well**: applying a coordinate's transformation to a single **`POINT`** can be considered a ***light-weight*** op;
  but transforming (*many and many times …*) some complex **`LINESTRING`** or **`POLYGON`** is much more an ***heavy-weight*** op.
  So we'll smartly use **`ST_Envelope()`** in order to strongly simplify any geometry requiring transformation.

All right, now the main framework of the complex query is absolutely clear:

- the first **`JOIN`** will include into the result-set any Populated Place falling within **20 Km** from the railway line.
- any other **`LEFT JOIN`** will then test decreasing distances, accordingly to the imposed Class boundaries.
- and each **LEFT JOIN** carefully checks if the Populated Place ID is the same of the previous successfully identified Class, as in: **`pp_d.id = pp_c.id`**
- each time one such **`LEFT JOIN`** will fail, then corresponding **`NULL`**-values will be inserted into the result-set.

```
SELECT rw.name AS Railway,
  pp_e.name AS PopulatedPlace,
  (ST_Distance(rw.geometry,
    Transform(pp_a.geometry, 23032)) <= 1000.0)
      AS "A class [< 1Km]",
...
```

Just a latest element to be shortly explained:

- consider a distance of e.g. **3.8 Km**: this requires inclusion in **C-class**. But this condition satisfies **`LEFT JOIN`** matching criteria for **D-class** and **E-class** as well.
- we have obviously to perform a further check.
- happily SQL is a really smart language. Any selected column to be returned into the result-set may represent any appropriate arbitrary expression.
- and in SQL a logical expression evaluates as:
  - **0** [*FALSE*]
  - **1** [*TRUE*]
  - or **NULL**, if anyone of the evaluated operands was **NULL**.
- and that's really all.

You can now play by yourself, performing further tests on this query.
i.e you can add some smart **`ORDER BY`** or **`WHERE`** clause and so on: that's really easy now, isn't ?

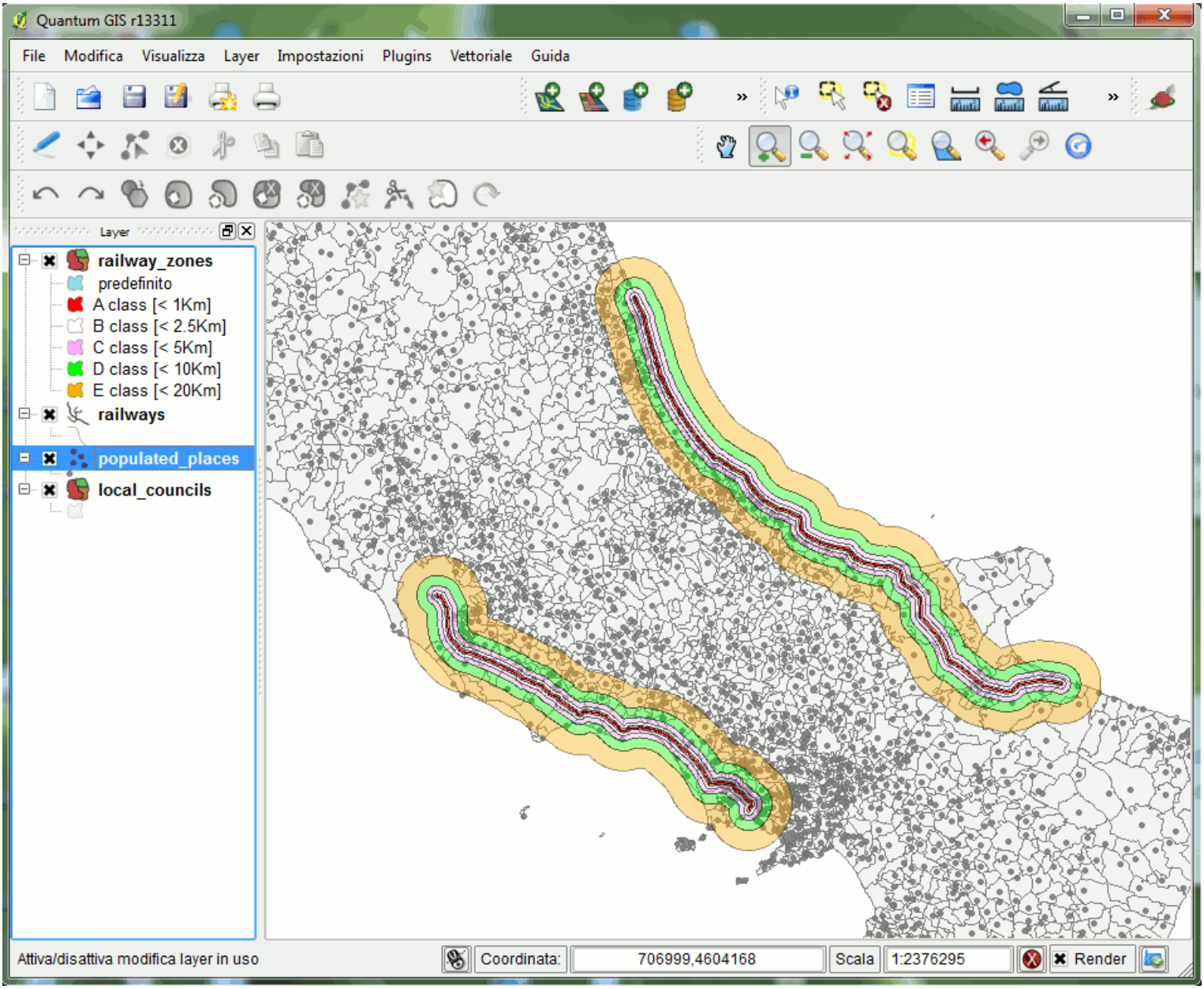
Recipe #18
Railway Zones as Buffers

2011 January 28

The problem

This is like a kind of *visual* conclusion of the latest exercise. The problem now is:

- create an appropriate **Map Layer** representing **A**, **B**, **C**, **D** and **E**-class **zones** as previously defined.




```
CREATE TABLE railway_zones (
  id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  railway_name TEXT NOT NULL,
  zone_name TEXT NOT NULL);
```

```
SELECT AddGeometryColumn('railway_zones', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');
```

We'll start creating a new table:

- as usual, we'll first create the table, omitting any Geometry column
- and we'll then create the Geometry column in a second time, using **`AddGeometryColumn()`**; but you already known all this.
- placing this new table into the **23032** SRID [***ED50 UTM zone 32***] is an absolutely obvious choice: after all, the original railways table is into the same identical SRID
- declaring a MULTYPOLYGON Geometry type is less obvious: but we'll see later why this is required.

```
INSERT INTO railway_zones
  (id, railway_name, zone_name, geometry)
SELECT NULL, name, 'A class [< 1Km]',
  CastToMultiPolygon(
    ST_Buffer(geometry, 1000.0))
FROM railways;
```

There is very little interest in this **`INSERT INTO ... SELECT ...`** statement (*again, you already known all this*).
Except for the following topic:

- we'll use **`ST_Buffer()`** to create a **`POLYGON`** corresponding to the **1Km A-class** zone.
- Please note: is not at all easy guessing the exact type of any Geometry created by **`ST_Buffer()`**;
  sometimes this function will create a **`POLYGON`**, but other times a **`MULTYPOLYGON`** may be created
  (*this depends on the exact shape of the input line, and obviously the given buffer radius has a strong influence as well*).
- so, in order to avoid any possible type inconsistency we defined a **`MULTIPOLYGON`** Geometry for this table.
- and we are now ***forcing*** the type by calling the explicit type-casting function **`CastToMultiPolygon()`**

```
INSERT INTO railway_zones
  (id, railway_name, zone_name, geometry)
SELECT NULL, name, 'B class [< 2.5Km]',
  CastToMultiPolygon(
    ST_Difference(
      ST_Buffer(geometry, 2500.0),
      ST_Buffer(geometry, 1000.0)))
FROM railways;
```

```
INSERT INTO railway_zones
  (id, railway_name, zone_name, geometry)
SELECT NULL, name, 'C class [< 5Km]',
  CastToMultiPolygon(
    ST_Difference(
      ST_Buffer(geometry, 5000.0),
      ST_Buffer(geometry, 2500.0)))
FROM railways;
```

```
INSERT INTO railway_zones
  (id, railway_name, zone_name, geometry)
SELECT NULL, name, 'D class [< 10Km]',
  CastToMultiPolygon(
    ST_Difference(
      ST_Buffer(geometry, 10000.0),
      ST_Buffer(geometry, 5000.0)))
FROM railways;
```

```
INSERT INTO railway_zones
  (id, railway_name, zone_name, geometry)
SELECT NULL, name, 'E class [< 20Km]',
  CastToMultiPolygon(
    ST_Difference(
      ST_Buffer(geometry, 20000.0),
      ST_Buffer(geometry, 10000.0)))
FROM railways;
```

Creating any further zone isn't much more difficult.

- we'll simply use **`ST_Difference()`** in order to get the appropriate representation:
  in other words, we must create an interior ***hole*** for each zone, so to exclude any other other zone we've already created.

You can now perform a simple ***visual check*** using QGIS. And that's all.

---

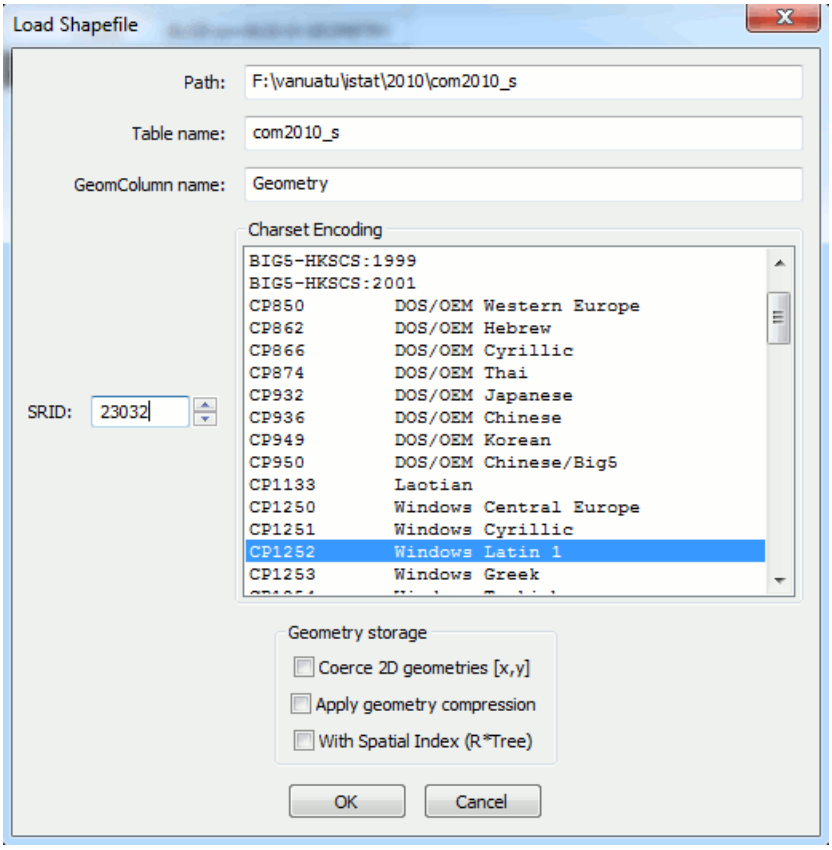
recipe #19
Merging Local Councils into Counties and so on ...

2011 January 28

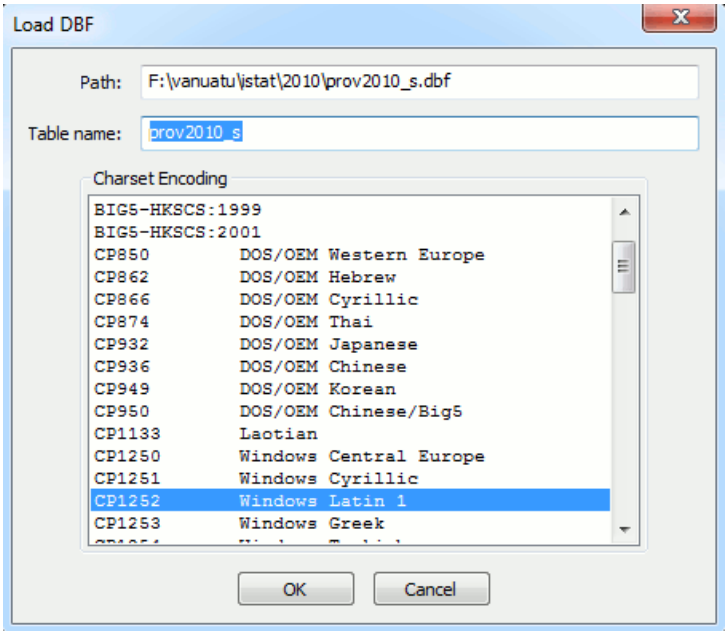
The problem

Local Councils, Counties and Regions follow a well defined ***order of hierarchy***.
For administrative purposes Italy is subdivided into Regions; Regions are subdivided into Counties; and Counties are subdivided into Local Councils.
Following the appropriate Spatial SQL procedures you can start from Local Councils geometries and then generate corresponding Counties geometries. And so on ...

Important notice: the ***ISTAT 2001 census*** dataset isn't well suited for this task, because it is plagued by several topology inconsistencies.
We'll use instead the latest ***ISTAT 2010*** dataset, presenting a much better quality and consistency.
http://www.istat.it/ambiente/cartografia/comuni2010.zip
http://www.istat.it/ambiente/cartografia/province2010.zip
http://www.istat.it/ambiente/cartografia/regioni2010.zip



You'll start creating a new DB; then using **`spatialite_gui`** you'll import the **`com2010_s`** ***shapefile***.



The next step is the one to load the **`prov2010_s`** dataset: yes, this one too actually is a *shapefile*.
But for your specific purposes you can ignore at all Counties Geometries.
(*generating all them by yourself is the specific task assigned to you this time, isn't ?*).
You can simply import the corresponding ***.DBF*** file, so to import any data but discarding and ignoring at all related Geometries.

Then you can import the Regions ***.DBF*** file from **`reg2010_s`**. exactly in the same way.

```
CREATE VIEW local_councils AS
SELECT c.cod_reg AS cod_reg,
  c.cod_pro AS cod_pro,
  c.cod_com AS cod_com,
  c.nome_com AS nome_com,
  p.nome_pro AS nome_pro,
  p.sigla AS sigla,
  r.nome_reg AS nome_reg,
  c.geometry AS geometry
FROM com2010_s AS c
JOIN prov2010_s AS p USING (cod_pro)
JOIN reg2010_s AS r USING(cod_reg);

SELECT * FROM local_councils;
```

cod_reg	cod_pro	cod_com	nome_com	nome_pro	sigla	nome_reg	geometry
1	1	1	Agliè	Torino	TO	PIEMONTE	BLOB sz=1117 GEOMETRY
1	1	2	Airasca	Torino	TO	PIEMONTE	BLOB sz=1149 GEOMETRY
1	1	3	Ala di Stura	Torino	TO	PIEMONTE	BLOB sz=1933 GEOMETRY
...	...	...	...	...	...	...	...

This will create the **`local_councils VIEW`**; this **`VIEW`** represents a ***nicely de-normalized flat table***, so to make any subsequent activity absolutely painless.

```
CREATE TABLE counties AS
SELECT cod_pro, nome_pro, sigla, cod_reg, nome_reg,
  ST_Union(geometry) AS geometry
FROM local_councils
GROUP BY cod_pro;

SELECT RecoverGeometryColumn('counties', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');
```

Now you'll create and populate the **`countries`** table:

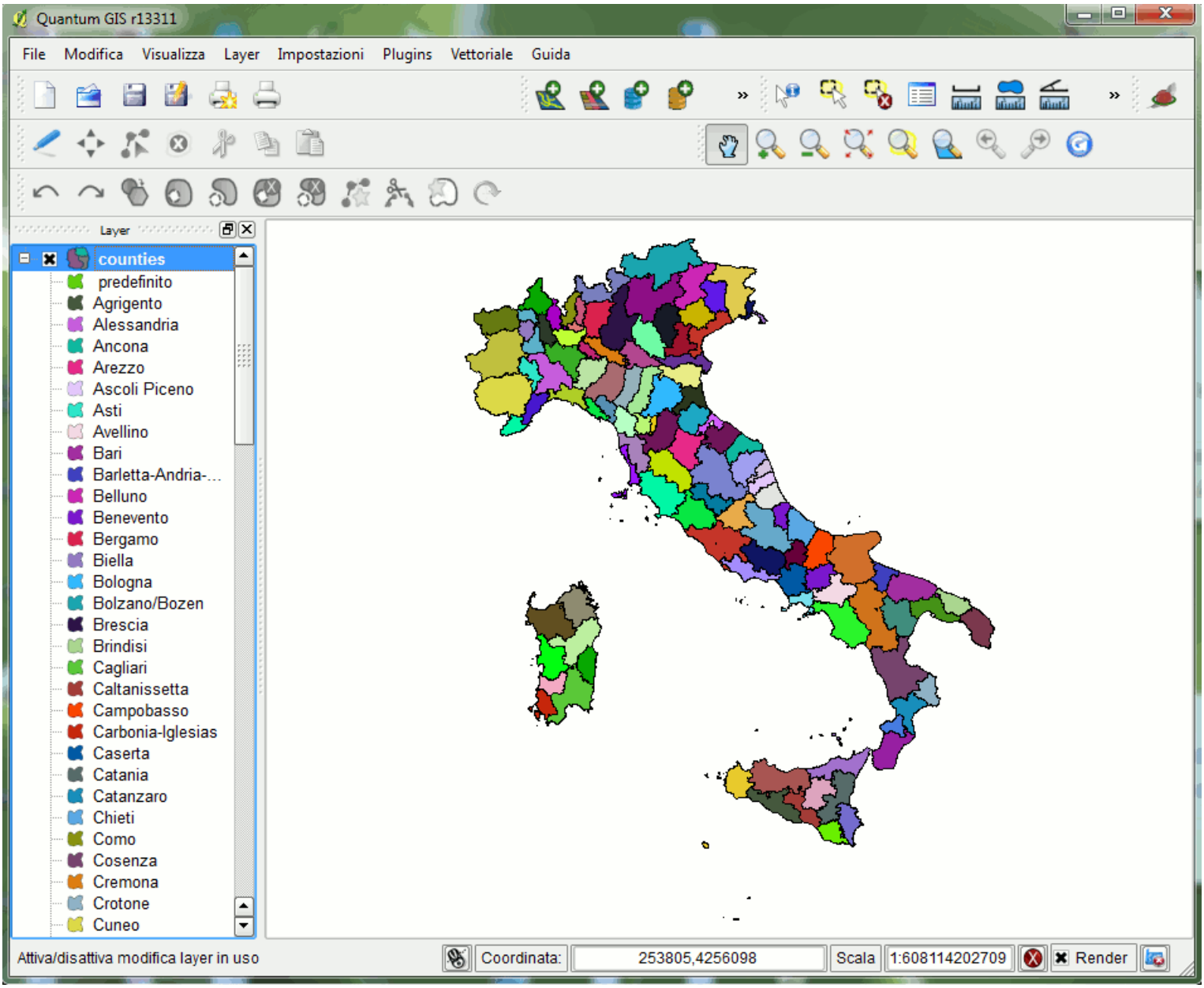
- the **`ST_Union()`** SQL Spatial function is used to aggregate / merge Geometries.
- by defining a **`GROUP BY cod_pro`** clause then **`ST_Union()`** will work as an *aggregate function*, thus effectively building the Geometry representation corresponding to each single County.
- **please note well**: you must absolutely call **`RecoverGeometryColumn()`** so to properly register the **`counties.geometry`** column into the **`geometry_columns`** *metadata* table.

```
SELECT * FROM counties;
```

cod_pro	nome_pro	sigla	cod_reg	nome_reg	geometry

1	Torino	TO	1	PIEMONTE	BLOB sz=36337 GEOMETRY
2	Vercelli	VC	1	PIEMONTE	BLOB sz=27357 GEOMETRY
3	Novara	NO	1	PIEMONTE	BLOB sz=15341 GEOMETRY
...	...	...	...	...	...

Just a quick check ...



And then you are ready to display the **`counties`** map layer using QGIS.

```
CREATE TABLE regions (
  cod_reg INTEGER NOT NULL PRIMARY KEY,
  nome_reg TEXT NOT NULL);

SELECT AddGeometryColumn('regions', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');

INSERT INTO regions (cod_reg, nome_reg, geometry)
SELECT cod_reg, nome_reg, ST_Union(geometry)
FROM counties
GROUP BY cod_reg;
```

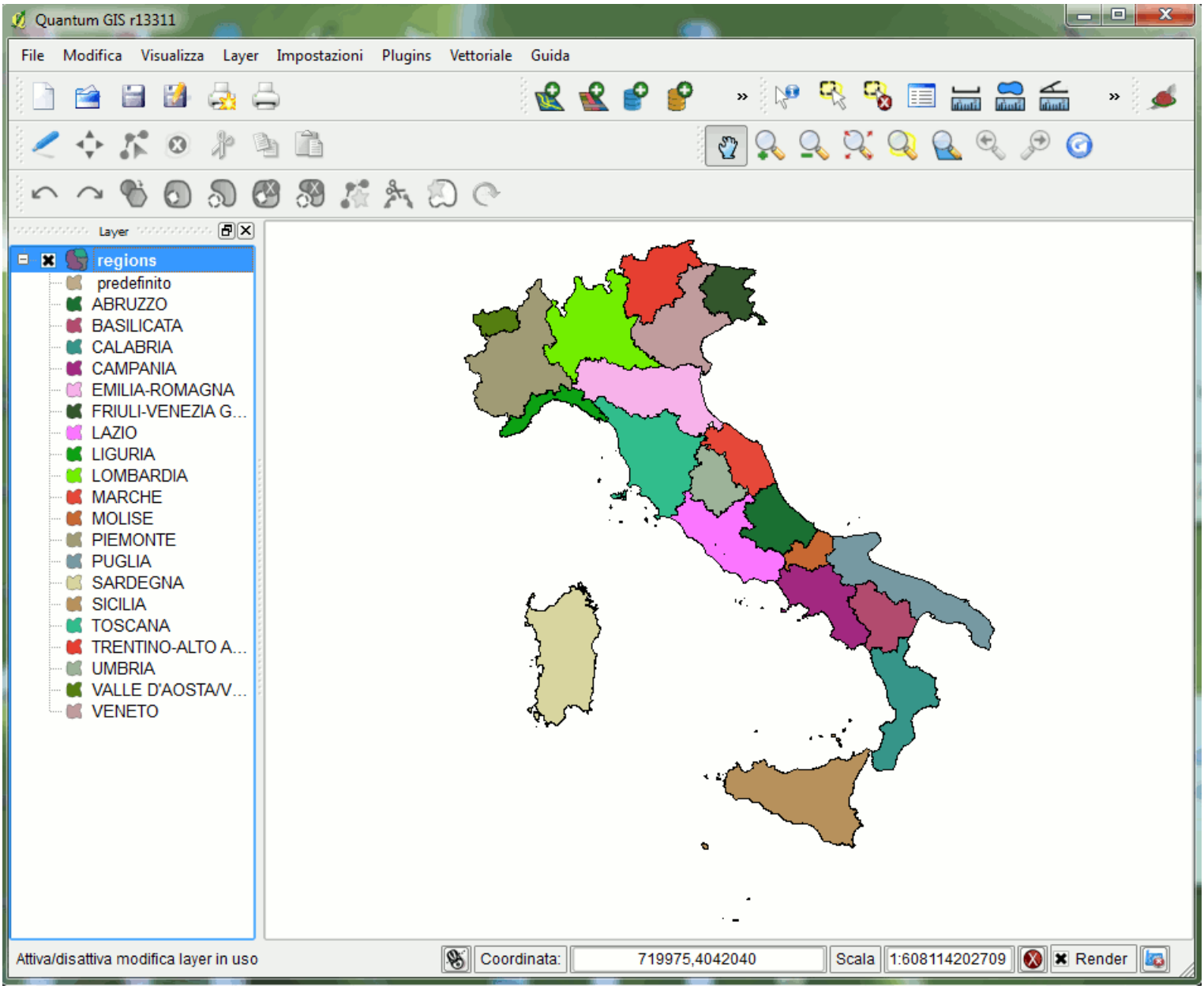
Now you'll create and populate the **`regions`** table:

- as in the previous step you'll use **`ST_Union()`** and **`GROUP BY`** to aggregate **`regions`** Geometries.
- **please note well**: in this example you have explicitly created the **`regions`** table, then using **`AddGeometryColumn()`** so to create the **`regions.geometry`** column.
  And finally you have used **`INSERT INTO ... (...) SELECT ...`** in order to populate the table.
  The procedure is different, but the final result is exactly the same one as in the previous example.

```
SELECT * FROM regions;
```

cod_reg	nome_reg	geometry
1	PIEMONTE	BLOB sz=75349 GEOMETRY
2	VALLE D'AOSTA/VALLÉE D'AOSTE	BLOB sz=18909 GEOMETRY
3	LOMBARDIA	BLOB sz=83084 GEOMETRY
...	...	...

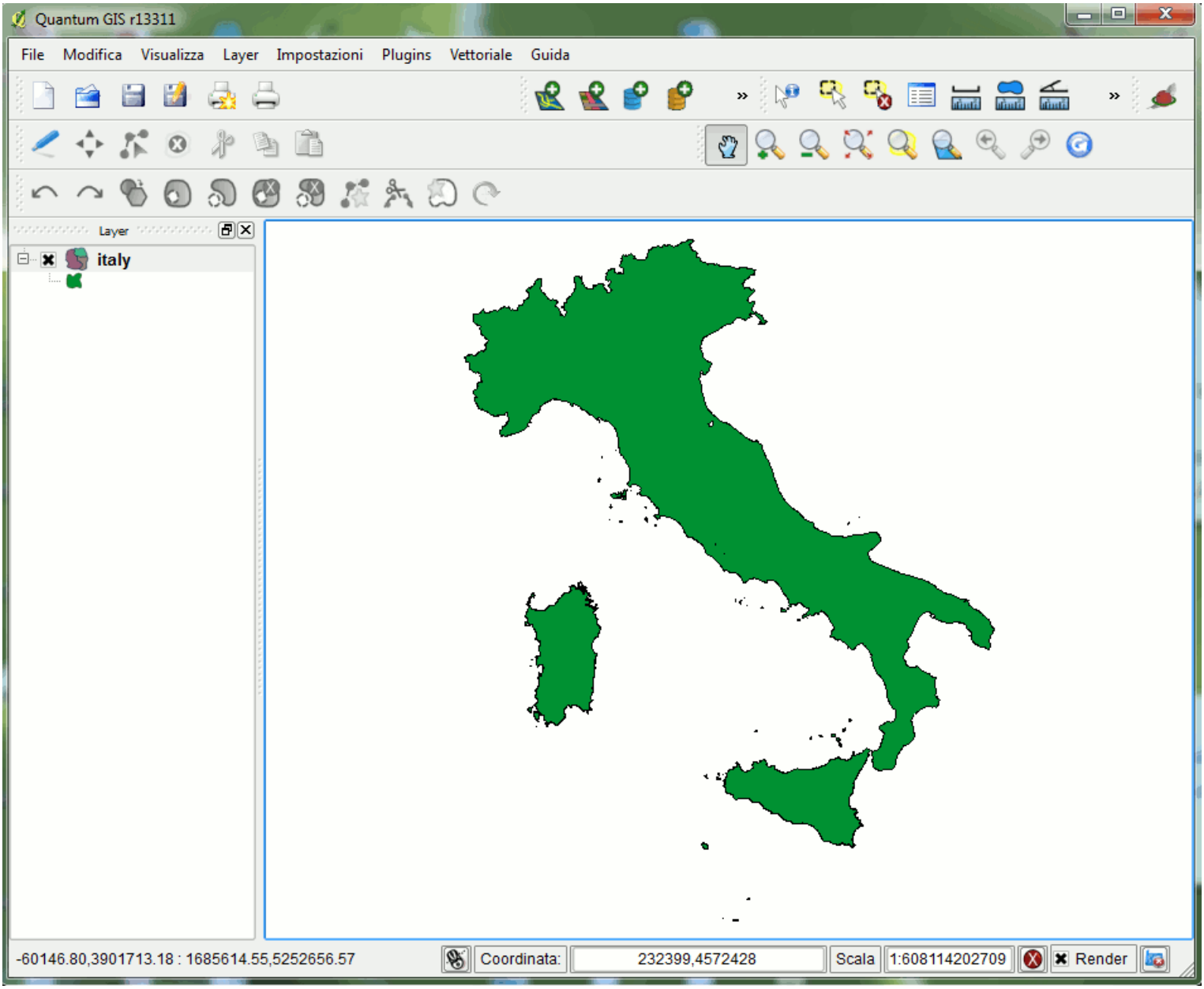
Just a quick check ...



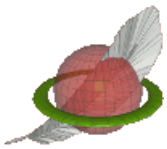
And then you can display the **regions** map layer using QGIS.

```
CREATE TABLE italy AS
SELECT 'Italy' AS country,
  ST_Union(geometry) AS geometry
FROM regions;

SELECT RecoverGeometryColumn('italy', 'geometry',
  23032, 'MULTIPOLYGON', 'XY');
```

As a final step you can now create the **italy** table representing the whole Italian Republic international boundaries.



Then you can display the **`italy`** map layer using QGIS … and that's all.



recipe #20
Spatial Views

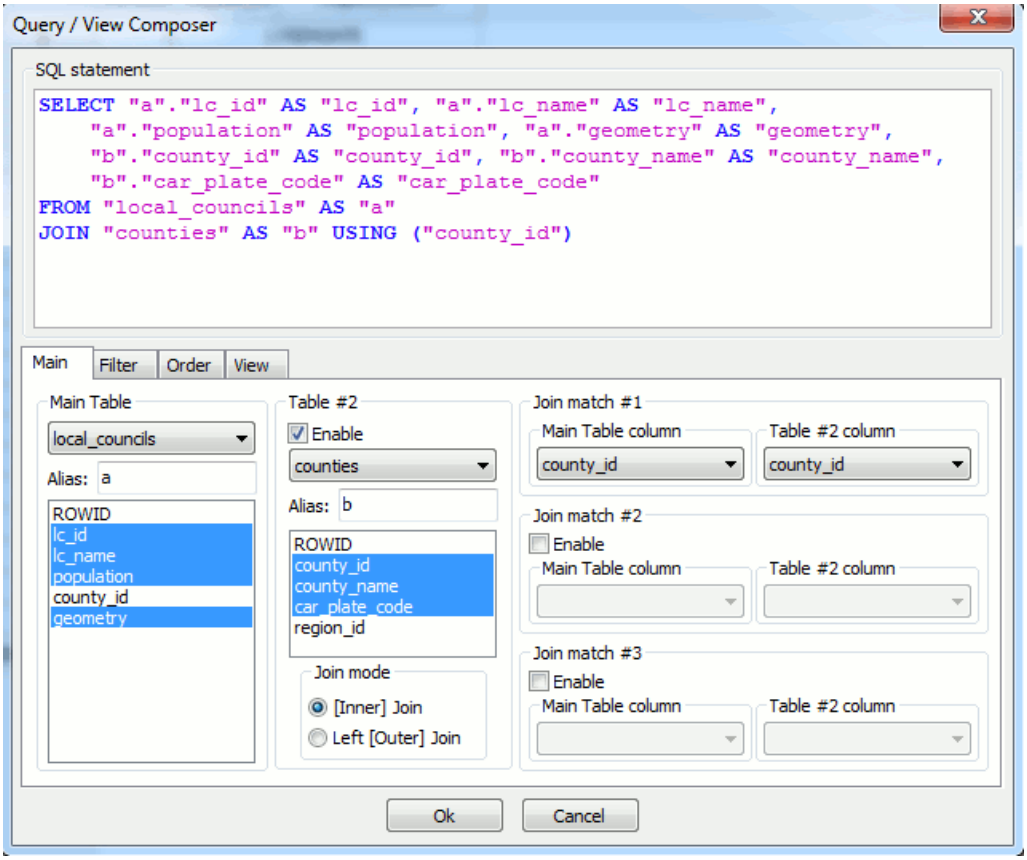
2011 January 28

SpatiaLite supports Spatial Views: any properly defined Spatial View can then be used as any other ***map layer***, i.e. can be displayed using QGIS .

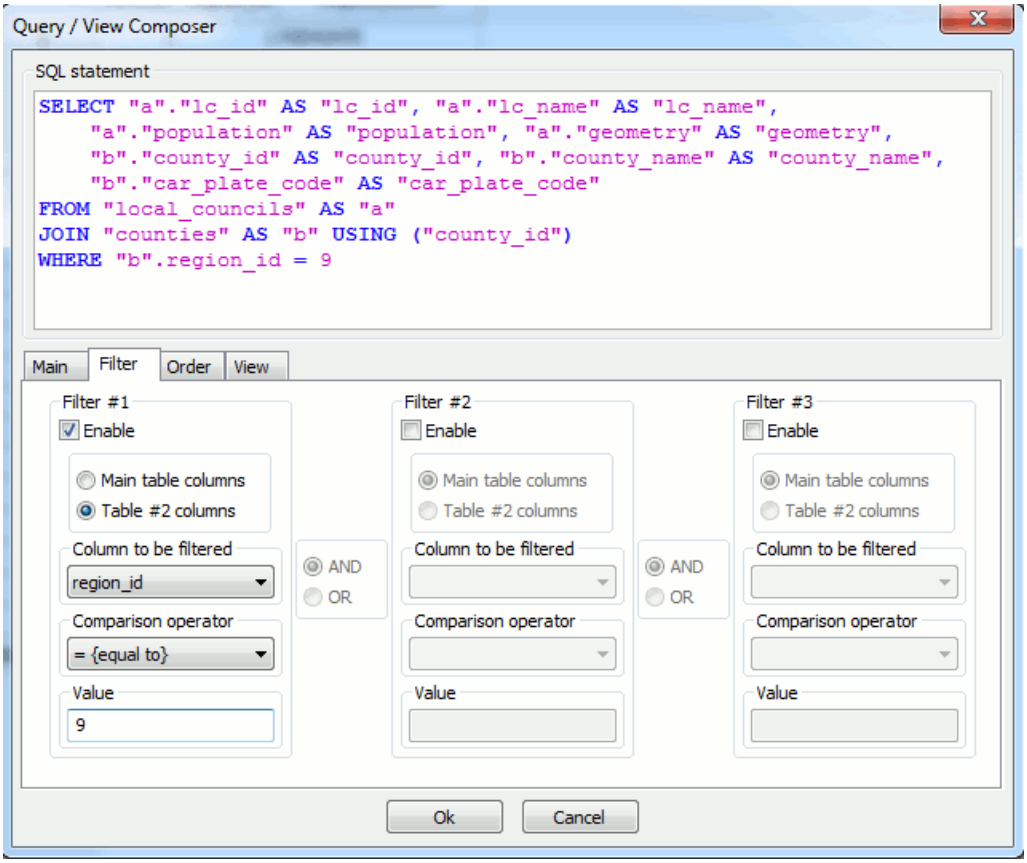
Please note: any SQLite **`VIEW`** can only be accessed in read-mode (**`SELECT`**);
and obviously such limitation applies to any Spatial View as well (no **`INSERT`**, **`DELETE`** or **`UPDATE`** are supported).

Using the query composer tool

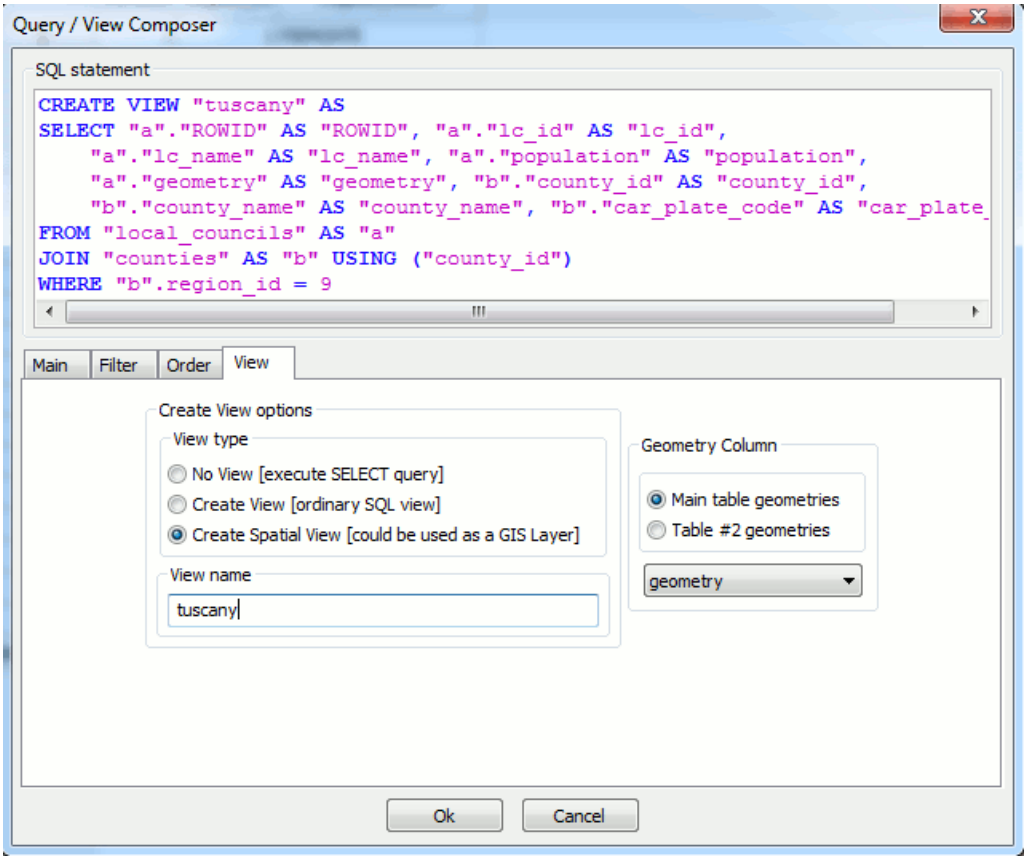
`spatialite_gui` supports a *query composer tool*; in this first example we'll use exactly this one.



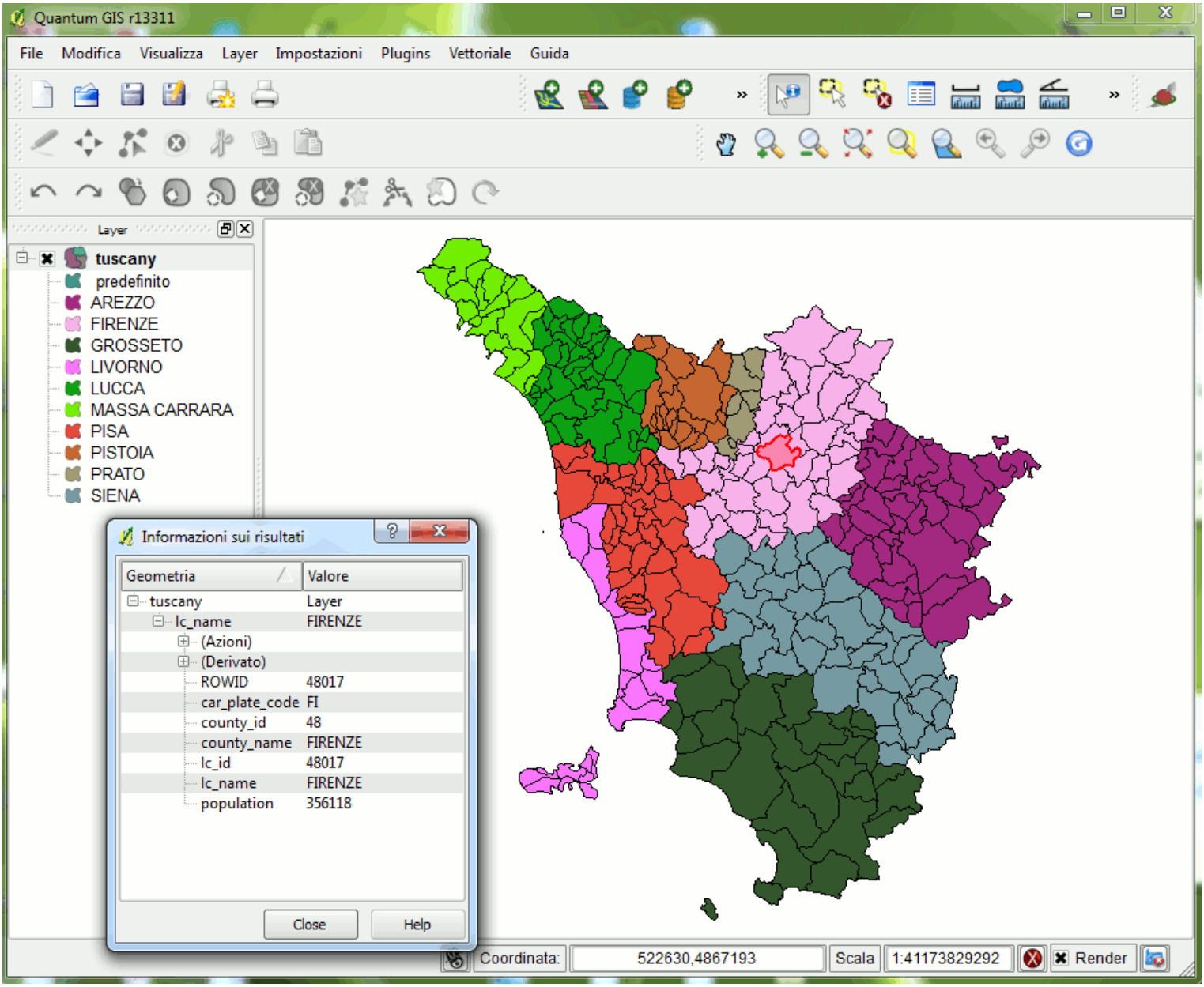
Step 1: selecting the required tables and columns, and defining the corresponding **`JOIN`** condition.
In this first example we'll **`JOIN`** the **`local_councils`** and the **`counties`** tables.



Step 2: now we'll set an appropriate ***filter*** clause;
in this case only **`local_councils`** and **`counties`** belonging to Tuscany Region (**`region_id = 9`**) will be extracted.



Step 3: and finally we'll set an appropriate **`VIEW`** name: during this latest phase we'll select the Geometry column corresponding to this **`VIEW`**.



We are now able to display this Spatial View using QGIS (an appropriate ***thematic rendering*** was applied so to evidentiate Counties).

Hand-writing your own Spatial VIEW

```
CREATE VIEW italy AS
SELECT lc.ROWID AS ROWID,
  lc.lc_id AS lc_id,
  lc.lc_name AS lc_name,
  lc.population AS population,
  lc.geometry AS geometry,
  c.county_id AS county_id,
  c.county_name AS county_name,
  c.car_plate_code AS car_plate_code,
  r.region_id AS region_id,
  r.region_name AS region_name
FROM local_councils AS lc
JOIN counties AS c ON (lc.county_id = c.county_id)
JOIN regions AS r ON (c.region_id = r.region_id);
```

You are not obligatorily compelled to use the *query composer tool*.
You are absolutely free to define any arbitrary VIEW to be used as a Spatial View.

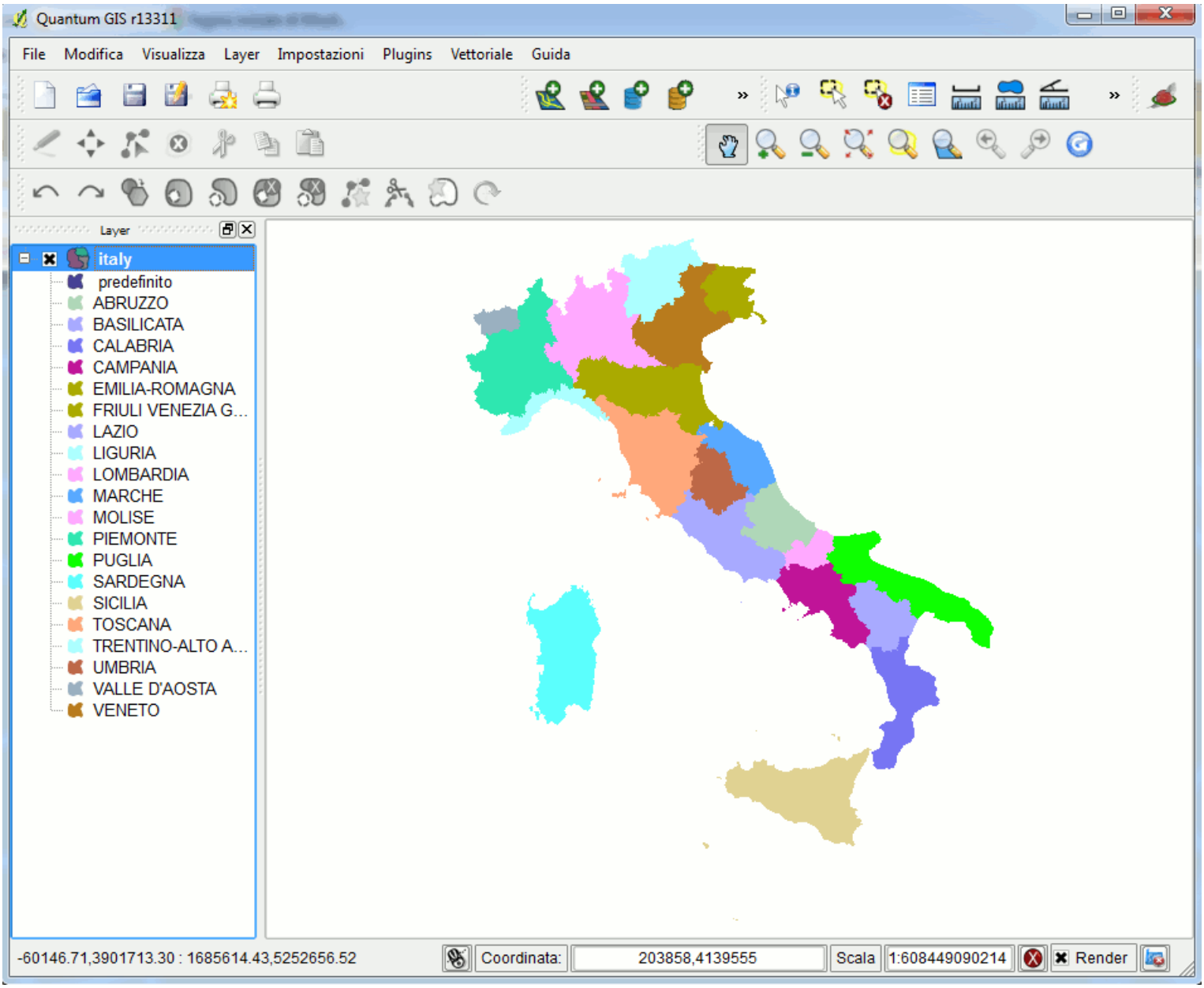
```
INSERT INTO views_geometry_columns
    (view_name, view_geometry, view_rowid, f_table_name, f_geometry_column)
  VALUES ('italy', 'geometry', 'ROWID', 'local_councils', 'geometry');
```

Anyway you must register this **`VIEW`** into the **`views_geometry_columns`**, so to make it become a ***real Spatial View***.

```
SELECT * FROM views_geometry_columns;
```


view_name	view_geometry	view_rowid	f_table_name	f_geometry_column
tuscany	geometry	ROWID	local_councils	geometry
italy	geometry	ROWID	local_councils	geometry

Just a simple check ...



And finally we can display this Spatial View using QGIS (an appropriate thematic rendering was applied so to evidentiate Regions).

Fine dining experience: Chez Dijkstra

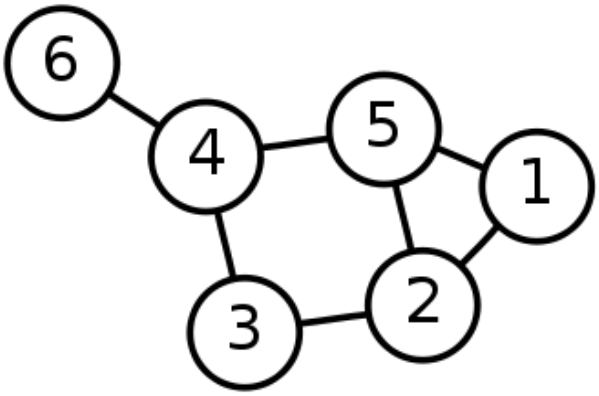
2011 January 28

SpatiaLite supports an internal **routing** module called `VirtualNetwork`
Starting from an arbitrary **network** this module allows to identify **shortest path** connections using simple SQL queries.
The `VirtualNetwork` module supports sophisticated and highly optimized algorithms, so it's really fast and very efficient even using huge sized networks.

Network foundations

You cannot assume that any generic *road layer* corresponds to a network.
A ***real network*** must satisfy several specific prerequisites, i.e. it has to be a **graph**.

Graph theory is a wide and complex branch of mathematics;
if you are interested in this, here you can get some further details:
Graph Theory
Shortest Path Problem
Dijkstra's Algorithm
A* Algorithm



Very shortly explained:

- a **network** is a collection of **arcs**
- each single arc connects two **nodes**
- each arc has an unique **direction**:
  i.e. the arc going from **A**-node to **B**-node is not necessarily the same one going from **B** to **A**
- each arc has a well known **cost** (e.g. *length, travel time, capacity*, ...)
- both arcs and nodes must expose some explicitly defined **unique identifier**.
- geometries of arcs and nodes must satisfy a strong **topological consistency**.

Starting from a **network** *aka* **graph** both **Dijkstra's** and **A*** algorithms can then identify the **shortest path** (*minimal cost connection*) connecting any arbitrary couple of nodes.

There are several sources distributing network-like data.
One of the most renowned and widely used is **OSM** [*Open Street Map*], a completely free worldwide dataset.

There are several download sites distributing OSM; just to mention the main ones:

- http://www.openstreetmap.org/
- http://download.geofabrik.de/osm/
- http://downloads.cloudmade.com/

Anyway in the following example we'll download the required OSM dataset from: www.gfoss.it
Most precisely we'll download the **`TOSCANA.osm.bz2`** dataset.

Step 1: you must uncompress the OSM dataset.
This file is compressed using the **bzip2** algorithm, widely supported by many open source tools.
e.g. you can use **7-zip** to unzip this file. www.7-zip.org

Step 2: any OSM dataset simply is an **XML** file
(*you can open this file using any ordinary text editor at your choice*).
SpatiaLite supports a specific CLI tool allowing to load an OSM dataset into a DB: **`spatialite_osm_net`**

```
>spatialite_osm_net -o TOSCANA.osm -d tuscany.sqlite -T tuscany -m
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
using IN-MEMORY database
Loading OSM nodes ... wait please ...
  Loaded 1642867 OSM nodes
Verifying OSM ways ... wait please ...
  Verified 60893 OSM ways
Disambiguating OSM nodes ... wait please ...
  Found 40 duplicate OSM nodes - fixed !!!
Loading network ARCs ... wait please ...
  Loaded 121373 network ARCs
Dropping temporary table 'osm_tmp_nodes' ... wait please ...
  Dropped table 'osm_tmp_nodes'
Dropping index 'from_to' ... wait please ...
  Dropped index 'from_to'
exporting IN_MEMORY database ... wait please ...
  IN_MEMORY database succesfully exported
VACUUMing the DB ... wait please ...
  All done: OSM graph was succesfully loaded
>
```

Very briefly explained:

- **`-o TOSCANA.osm`** selects the input OSM dataset to be loaded.
- **`-d tuscany.sqlite`** selects the output DB to be created and populated.
- **`-T tuscany`** will create the Geometry Table storing the OSM dataset
- **`-m`** an *in-memory database* will be used, so to perform data import in the shortest time.

```
SELECT *
FROM tuscany;
```

id	osm_id	class	node_from	node_to	name	oneway_from_to	oneway_to_from	length	cost	geometry
...	...	...	...	...	...	...	...	...	...	...
2393	8079944	tertiary	659024545	659024546	Via Cavour	1	1	7.468047	0.537699	BLOB sz=80 GEOMETRY
2394	8079944	tertiary	659024546	156643876	Via Cavour	1	1	12.009911	0.864714	BLOB sz=96 GEOMETRY
2395	8083989	motorway	31527668	319386487	Autostrada del Sole	1	0	424.174893	13.882087	BLOB sz=80 GEOMETRY
2396	8083990	motorway	31527665	31527668	Autostrada del Sole	1	0	130.545183	4.272388	BLOB sz=112 GEOMETRY
...	...	...	...	...	...	...	...	...	...	...

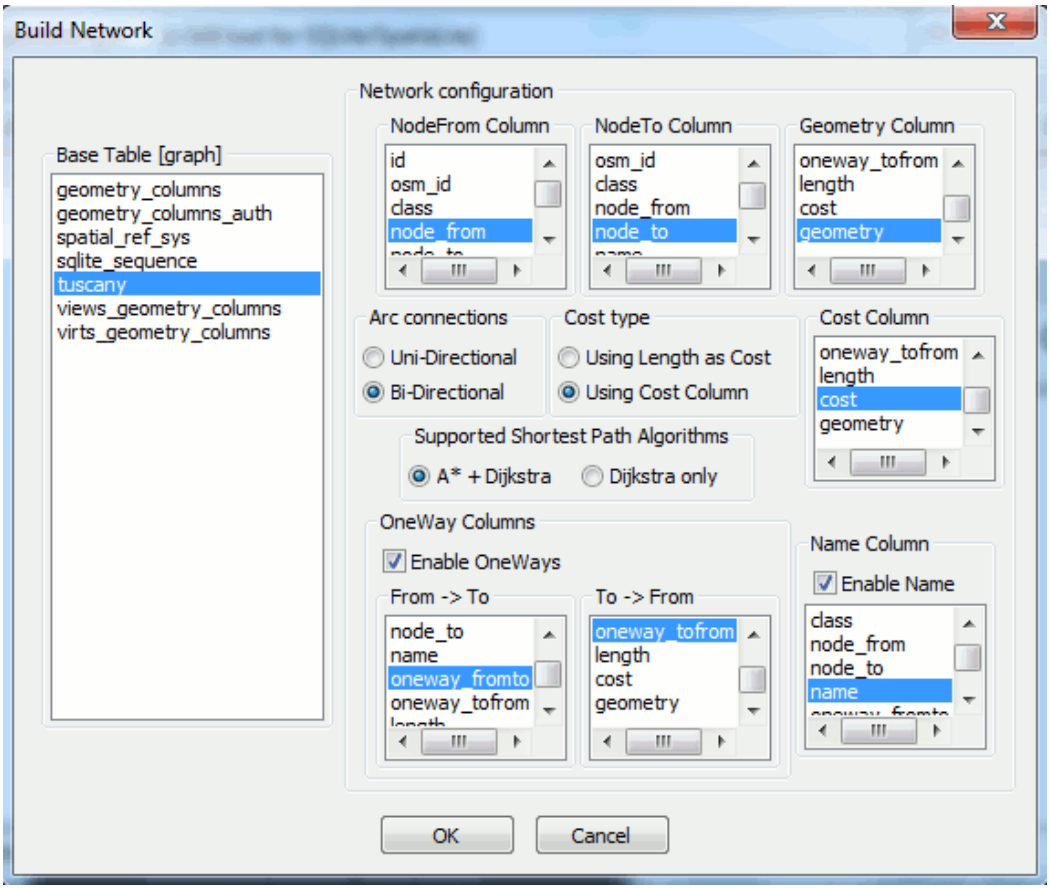
Just a quick check:

- a single **`Tuscany`** table exists into the DB created by **`spatialite_osm_net`**
- each row in this table corresponds to a single **network arc**
- the nodes connected by each arc are identified by **`node_from`** and **`node_to`**

- **`oneway_from_to`** and **`oneway_to_from`** determine if the arc can be walked in both directions or not.
- **`length`** is the geometric length of the arc (measured in **meters**).
- **`cost`** is the estimated travel time (expressed in **seconds**).
- **`geometry`** is the **`LINESTRING`** representation corresponding to the arc.

Please note #1: there is no separate representation for **nodes**, simply because they can be indirectly retrieved starting from the corresponding arcs.

Please note #2: this one surely is a ***real network***, but in this form cannot yet support **routing queries**.
A further step is still required, i.e. creating a **`VirtualNetwork`** table.



We'll use **`spatialite_gui`** to create the VirtualNetwork table.
Anyway the same operation is supported as well by the **`spatialite_network`** CLI tool
(*and this CLI tool supports an extended diagnostic capability, useful to identify any eventual problem*).

```
SELECT *
FROM tuscany_net
WHERE NodeFrom = 267209305
  AND NodeTo = 267209702;
```

Algorithm	ArcRowid	NodeFrom	NodeTo	Cost	Geometry	Name
Dijkstra	NULL	267209305	267209702	79.253170	BLOB sz=272 GEOMETRY	NULL
Dijkstra	11815	267209305	250254381	11.170037	NULL	Via Guelfa
Dijkstra	11816	250254381	250254382	8.583739	NULL	Via Guelfa
Dijkstra	11817	250254382	250254383	12.465016	NULL	Via Guelfa
Dijkstra	16344	250254383	256636073	15.638407	NULL	Via Cavour
Dijkstra	67535	256636073	270862435	3.147105	NULL	Piazza San Marco
Dijkstra	25104	270862435	271344268	5.175379	NULL	Piazza San Marco
Dijkstra	25105	271344268	82591712	3.188657	NULL	Piazza San Marco
Dijkstra	11802	82591712	267209666	4.978328	NULL	Piazza San Marco
Dijkstra	20773	267209666	267209702	14.906501	NULL	Via Giorgio La Pira

And finally you can now test your first routing query:

- you simply have to set the **`WHERE NodeFrom = ... AND NodeTo = ...`** clause.
- and a result-set representing the **shortest path** solution will be returned.
- the *first row* of this result-set summarizes the whole path, and contains the corresponding geometry.
- any *subsequent row* represents a single arc to be traversed, following the appropriate sequence, so to go from origin to destination.

```
UPDATE tuscany_net SET Algorithm = 'A*';

UPDATE tuscany_net SET Algorithm = 'Dijkstra';
```

SpatiaLite's **`VirtualNetwork`** tables support two alternative algorithms:

- ***Dijkstra's shortest path*** is a *classic* routing algorithm, based on thorough mathematical assumptions, and will surely identify the optimal solution.
- **A*** is an alternative algorithm based on *heuristic* assumptions:
  it is usually faster than Dijkstra's, but under some odd condition may eventually fail, or may return a sub-optimal solution.
- anyway switching from the one to the other is really simple.
- using the Dijksta's algorithm is the default selection.

A **`VirtualNetwork`** table simply represents a *staticized snapshot* of the underlying network.
This allows to adopt an highly efficient binary representation (*in other words, allows to produce solutions in a very quick time*), but obviously doesn't supports dynamic changes.

Each time the underlying network changes the corresponding **`VirtualNetwork`** must be **`DROP`**ped and then created again, so to correctly reflect the latest network state.

In many cases this isn't an issue at all: but on some *highly dynamic scenario* this may be a big annoyance.
Be well conscious of this limitation.

System level performace hints

2011 January 28

We have examined since now several optimization related topics: but all this mainly was to be intended as "*smartly writing well designed queries*".

Although defining a properly planned SQL query surely represents the main factor to achieve optimal performances, this isn't enough.
A second level of performance optimization (*fine tuning*) exist, i.e. the one concerning interactions between the DBMS and the underlying ***Operating System*** / ***File System***.

DB pages / page cache

Any SQLite DB simply is a single monolithic file: any data and related info is stored within this files.
As in many others DBMS, disk space isn't allocated at random, but is properly structured:
the *atomic* allocation unit is defined as a **page**, so a DB file simply is a well organized collection of ***pages***.
All pages within the same DB must have the same identical size (typically **`1KB`** i.e. **`1024 bytes`**):

- adopting a bigger page size may actually reduce the I/O traffic, but may impose to waste a significant amount of unused space.
- adopting a smaller page size is strongly discouraged, because will surely imply a much more sustained I/O traffic.
- so the *default* page size of **`1KB`** represents a *mean case* well fitted for the vast majority of real world situations.

Reading and writing from disk a single page at each time surely isn't an efficient process;
so SQLite maintains an internal ***page cache*** (*stored in RAM*), supporting fast access to the most often accessed pages.
Quite intuitively, adopting a bigger page cache can strongly reduce the overall I/O traffic;
and consequently an higher throughput can be achieved.

By default SQLite adopts a very conservative approach, so to require a light-weight memory footprint;
the initial page cache will simply store **`2000 pages`** (corresponding to a total allocation of only **`20MB`**).

But a so small default page cache surely isn't enough to properly support an ***huge DB***, (*may be one ranging in the **many-GB** size*);
this will easily become a real ***bottleneck***, causing very poor global performances.

`PRAGMA page_size;`
`1024`
`PRAGMA page_count;`
`31850`
`PRAGMA freelist_count;`
`12326`

You can use several **`PRAGMA`**s to check the page status for the currently connected DB:

- **PRAGMA page_size;** will report the currently set page size.
- **PRAGMA page_count;** will report the total number of allocated pages.
- **PRAGMA freelist_count;** will report the total number of unused pages.
  - please note: each time you perform lots of **DELETE**s or some **DROP** [**TABLE** | **INDEX**] statement, then several unused pages will be left in the DB.

PRAGMA page_size = 4096;
PRAGMA page_size;
1024

You can call a **PRAGMA page_size** so to set a different page size
(you must specify a ***power of two*** size argument, ranging from **512** to **65536**):

- anyway, the page size still remains unchanged.
- this is because a complete DB reorganization is required in order to make such change to actually materialize.

VACUUM;

Performing a **VACUUM** implies the following actions to be performed:

- the whole DB will be checked and completely rewritten from scratch.
- any structural change (*e.g. changing the current page size*) will now be applied.
- any unused page will be discarded, so the DB will be effectively compacted.
- please note: **VACUUM**ing a large DB may require a long time.

PRAGMA page_size;
4096
PRAGMA page_count;
5197
PRAGMA freelist_count;
0

Just a quick check: immediately after performing **VACUUM** the new page size has been effectively applied, and there are no unused pages at all.

PRAGMA cache_size;
1000
PRAGMA cache_size = 1000000;
PRAGMA cache_size;
1000000

You can use **PRAGMA cache_size** in order to query or set the page cache:

- **please note**: the size is measured as the *number of pages* to be stored into the cache:
  so the corresponding memory allocation (*in bytes*) will be **page_size * cache_size**
- requesting a bigger cache size usually implies better performances.
  Anyway carefully consider that:
  - an exaggeratedly big cache is completely useless: you'll simply waste a lot of precious RAM for nothing.
  - when the memory allocation required by the page cache exceed the amount of physically available RAM, performance will then be catastrophically impaired, because this will actually cause an enormous I/O traffic due to ***memory-to-disk swapping***.
  - on 32bit platforms a further limit exist: on such platforms any process cannot allocate more than 4GB

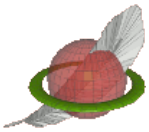
memory.
But for practical reasons this limit is more likely to be as low as 1.5GB.

Requesting a very generously (*but wisely*) dimensioned page cache usually will grant a great performance boost, most notably when you are processing a very large DB.

You can modify other important settings using the appropriate **PRAGMA**s supported by SQLite:

- **PRAGMA ignore_check_constraint** can be used to query, enable or disable **CHECK** constraints (*e.g. disabling check constraints is unsafe, but may be required during preliminary data loading*).
- **PRAGMA foreign_key** can be used to query, enable or disable **FOREIGN KEY** constraints (*and this too may be useful or required during preliminary data loading*).
- **PRAGMA journal_mode** can be used to query or set fine details about **TRANSACTION** journaling.

PRAGMA's implementation change from time to time, so you can usefully consult the appropriate SQLite documentation



Importing / Exporting Shapefiles (DBF, TXT ...)

2011 January 28

There are several *data formats* that are absolutely widespread in the GIS professional world.
All them are *open formats* (i.e. they are not specifically bounded to any specific proprietary software, they are not patent covered, and they are publicly documented).

Not at all surprisingly, such formats are universally supported by any GIS-related software, and can be safely used for data interchange purposes between different platforms and systems.

- the **ESRI Shapefile** format represents the *de-facto* universal format for GIS data exchange.
- the **DBF** format was introduced in the very early days of personal computing by **dBase** (*the first DBMS-like software to gain universal popularity*): a *DBF* file is included in every *Shapefile* but is quite common finding *naked DBF* files simply used to ship *flat tables*.
- the **TXT/CSV** format simply identifies any *structured text file* (usually, *tab separated values* or *comma separated values* are the most often found variants).

SpatiaLite supports all the above formats for import and/or export.

Please note well: other data formats are very popular and widespread. e.g. the following ones:

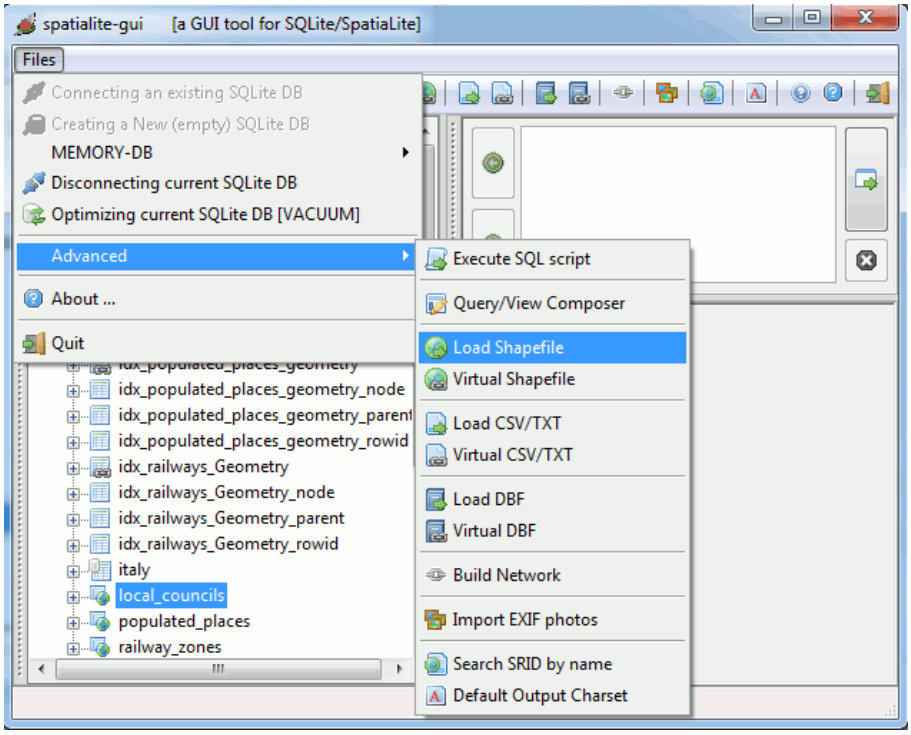
- Microsoft Excel spreadsheets (**.xls**)
- Microsoft Access database (**.mdb**)
- and many, many others.

All them are *closed (proprietary) formats*.
i.e. some specific proprietary software or operating system is strictly required, there is no publicly available documentation, and/or they are patent covered.
And all this easily explains why SpatiaLite (as many others open source packages) cannot support such *closed formats*.

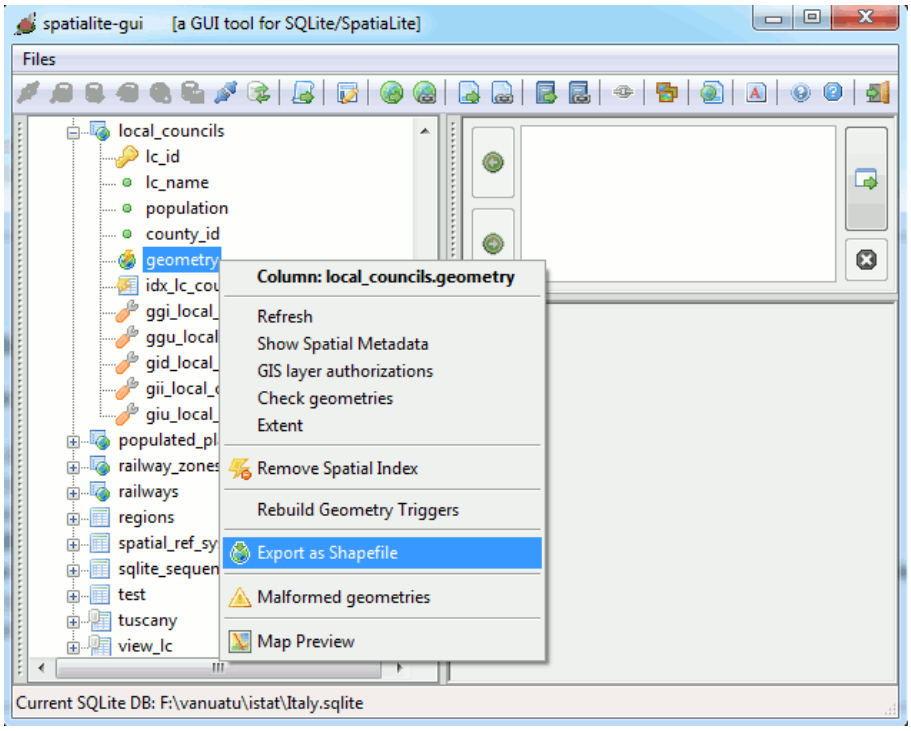
Shapefiles

SpatiaLite supports both import and export for Shapefiles:

- you can directly access any external Shapefile via SQL as a **VirtualShapefile** table
- you can import any Shapefile into a DB Table:
  - using **`spatialite_gui`** you'll find a **`Load Shapefile`** item in the main menu and in the toolbar.
  - using the **`spatialite`** CLI front-end you can use the **`.loadshp`** macro
  - or you can use the **`spatialite_tool`** shell command
- you can export any Geometry Table as a Shapefile:
  - using **`spatialite_gui`** you'll find an **`Export As Shapefile`** item on the context menu corresponding to any Geometry column within the main tree-view.
  - using the **`spatialite`** CLI front-end you can use the **`.dumpshp`** macro
  - or you can use the **`spatialite_tool`** shell command



`spatialite_gui`: Shapefile import



`spatialite_gui`: Shapefile export

```
> spatialite counties.sqlite
SpatiaLite version ..: 2.4.0-RC5 Supported Extensions:
        - 'VirtualShape'        [direct Shapefile access]
        - 'VirtualDbf'          [direct DBF access]
        - 'VirtualText'         [direct CSV/TXT access]
        - 'VirtualNetwork'      [Dijkstra shortest path]
        - 'RTree'               [Spatial Index - R*Tree]
        - 'MbrCache'            [Spatial Index - MBR cache]
        - 'VirtualFDO'          [FDO-OGR interoperability]
        - 'SpatiaLite'          [Spatial SQL - OGC]
PROJ.4 version ......: Rel. 4.7.1, 23 September 2009
GEOS version ........: 3.3.0-CAPI-1.7.0
SQLite version ......: 3.7.4
Enter ".help" for instructions
spatialite> .loadshp prov2010_s counties CP1252 23032
the SPATIAL_REF_SYS table already contains some row(s)
========
Loading shapefile at 'prov2010_s' into SQLite table 'counties'

BEGIN;
CREATE TABLE counties (
PK_UID INTEGER PRIMARY KEY AUTOINCREMENT,
"OBJECTID" INTEGER,
"COD_PRO" INTEGER,
"NOME_PRO" TEXT,
"SIGLA" TEXT);
SELECT AddGeometryColumn('counties', 'Geometry', 23032, 'MULTIPOLYGON', 'XY');
```

```
COMMIT;

Inserted 110 rows into 'counties' from SHAPEFILE
========
spatialite> .headers on
spatialite> SELECT * FROM counties LIMIT 5;
PK_UID|OBJECTID|COD_PRO|NOME_PRO|SIGLA|Geometry
1|1|1|Torino|TO|
2|2|2|Vercelli|VC|
3|3|3|Novara|NO|
4|4|4|Cuneo|CN|
5|5|5|Asti|AT|
spatialite>
```

spatialite CLI front-end: shapefile import

```
spatialite> .dumpshp counties Geometry exported_counties CP1252
========
Dumping SQLite table 'counties' into shapefile at 'exported_counties'

SELECT * FROM "counties" WHERE GeometryAliasType("Geometry") = 'POLYGON'
OR GeometryAliasType("Geometry") = 'MULTIPOLYGON' OR "Geometry" IS NULL;

Exported 110 rows into SHAPEFILE
========
spatialite> .quit

>
```

spatialite CLI front-end: shapefile export

```
> spatialite_tool -i -shp prov2010_s -d db.sqlite -t counties -c CP1252 -s 23032
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
Inserted 110 rows into 'counties' from 'prov2010_s.shp'

>
```

spatialite_tool shell command: shapefile import

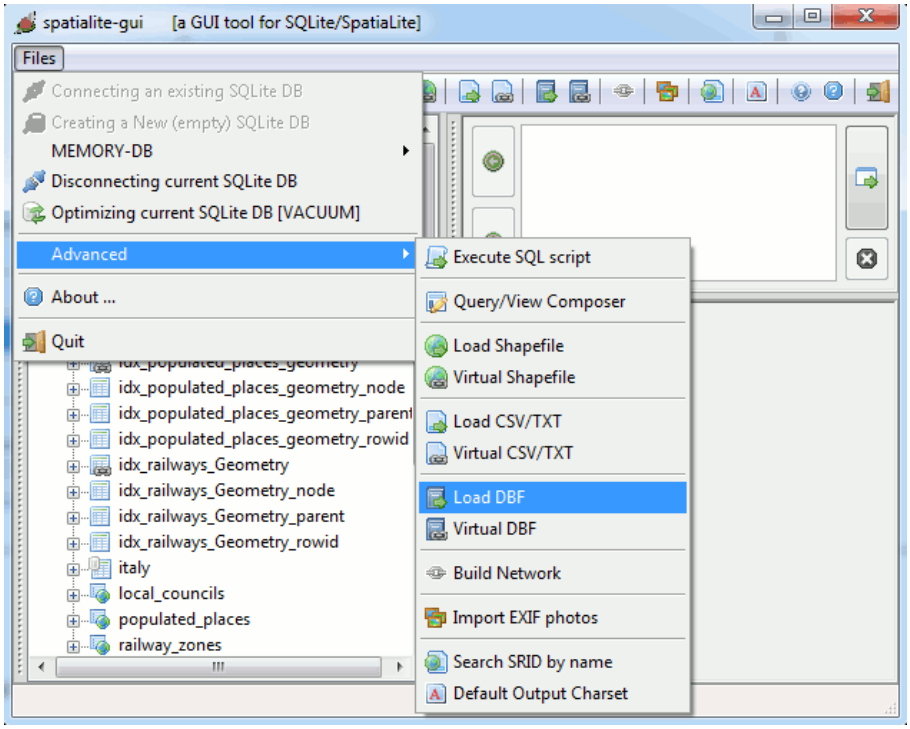
```
> spatialite_tool -e -shp exported_counties -d db.sqlite -t counties -g Geometry -c CP1252
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
Exported 110 rows into 'exported_counties.shp' from 'counties'

>
```

spatialite_tool shell command: shapefile export

DBF files

SpatiaLite simply supports import for DBF files:

- you can directly access any external DBF file via SQL as a **VirtualDbf** table
- you can import any DBF file into a DB Table:
  - using **spatialite_gui** you'll find a **Load DBF** item in the main menu and in the toolbar.
  - using the **spatialite** CLI front-end you can use the **.loaddbf** macro
  - or you can use the **spatialite_tool** CLI command



spatialite_gui: DBF import

```
> spatialite local_councils.sqlite
```

```
SpatiaLite version ..: 2.4.0-RC5 Supported Extensions:
        - 'VirtualShape'        [direct Shapefile access]
        - 'VirtualDbf'          [direct DBF access]
        - 'VirtualText'         [direct CSV/TXT access]
        - 'VirtualNetwork'      [Dijkstra shortest path]
        - 'RTree'               [Spatial Index - R*Tree]
        - 'MbrCache'            [Spatial Index - MBR cache]
        - 'VirtualFDO'          [FDO-OGR interoperability]
        - 'SpatiaLite'          [Spatial SQL - OGC]
PROJ.4 version ......: Rel. 4.7.1, 23 September 2009
GEOS version ........: 3.3.0-CAPI-1.7.0
SQLite version ......: 3.7.4
Enter ".help" for instructions
spatialite> .loaddbf com2010_s.dbf local_councils CP1252
========
Loading DBF at 'com2010_s.dbf' into SQLite table 'local_councils'

BEGIN;
CREATE TABLE local_councils (
PK_UID INTEGER PRIMARY KEY AUTOINCREMENT,
"OBJECTID" INTEGER,
"COD_REG" INTEGER,
"COD_PRO" INTEGER,
"COD_COM" INTEGER,
"PRO_COM" INTEGER,
"NOME_COM" TEXT,
"NOME_ITA" TEXT,
"NOME_TED" TEXT);
COMMIT;

Inserted 8094 rows into 'local_councils' from DBF ========
spatialite> .headers on
spatialite> SELECT * FROM local_councils LIMIT 5 OFFSET 5000;
PK_UID|OBJECTID|COD_REG|COD_PRO|COD_COM|PRO_COM|NOME_COM|NOME_ITA|NOME_TED
5001|4958|12|58|54|58054|Manziana|Manziana|
5002|4959|12|58|55|58055|Marano Equo|Marano Equo|
5003|4960|12|58|56|58056|Marcellina|Marcellina|
5004|4961|12|58|57|58057|Marino|Marino|
5005|4962|12|58|58|58058|Mazzano Romano|Mazzano Romano|
spatialite> .quit

>
```

`spatialite` CLI front-end: DBF import

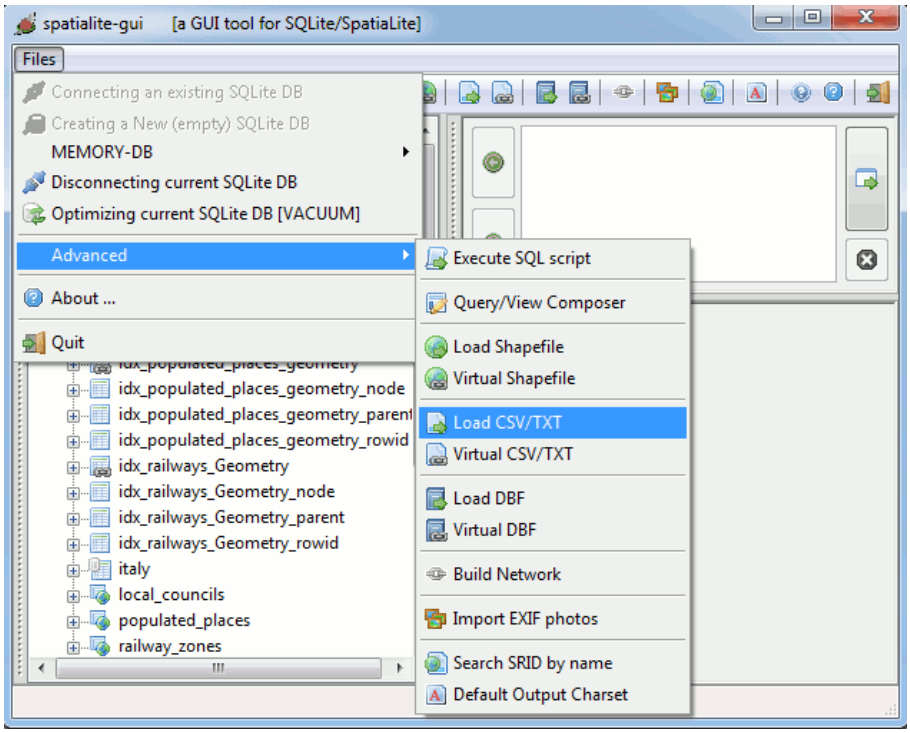
```
> spatialite_tool -i -dbf com2010_s -d db.sqlite -t local_councils -c CP1252
SQLite version: 3.7.4
SpatiaLite version: 2.4.0-RC5
Inserted 8094 rows into 'local_councils' from 'com2010_s.dbf'

>
```

`spatialite_tool` shell command: DBF import

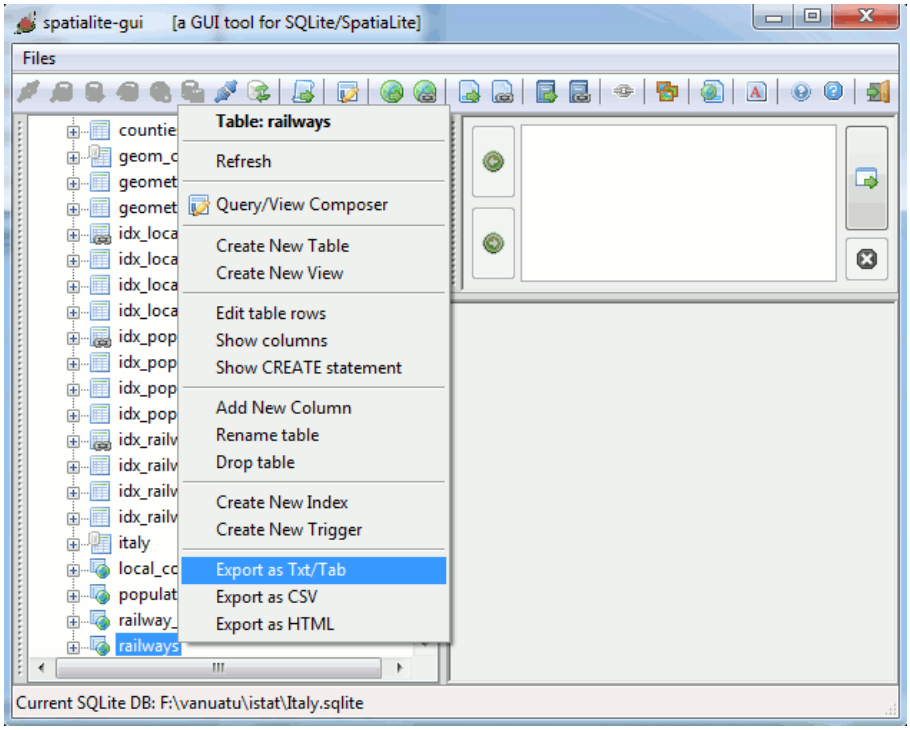
TXT/CSV files

SpatiaLite supports both import and export for TXT/CSV files:

- you can directly access any external TXT/CSV file via SQL as **VirtualText** table
- you can import any TXT/CSV file into a DB Table:
  - using **`spatialite_gui`** you'll find a **`Load TXT/CSV`** item in the main menu and in the toolbar.
- you can export any Table as a TXT/CSV file:
  - using **`spatialite_gui`** you'll find an **`Export As TXT/CSV`** item on the context menu corresponding to any Table within the main tree-view.



`spatialite_gui`: TXT/CSV import

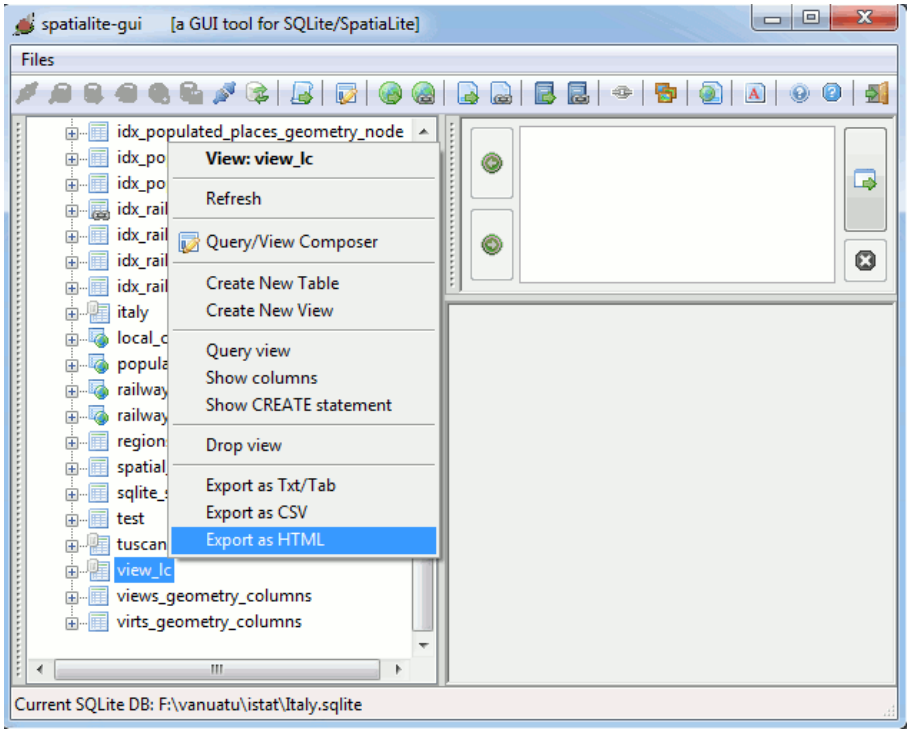


spatialite_gui: TXT/CSV export

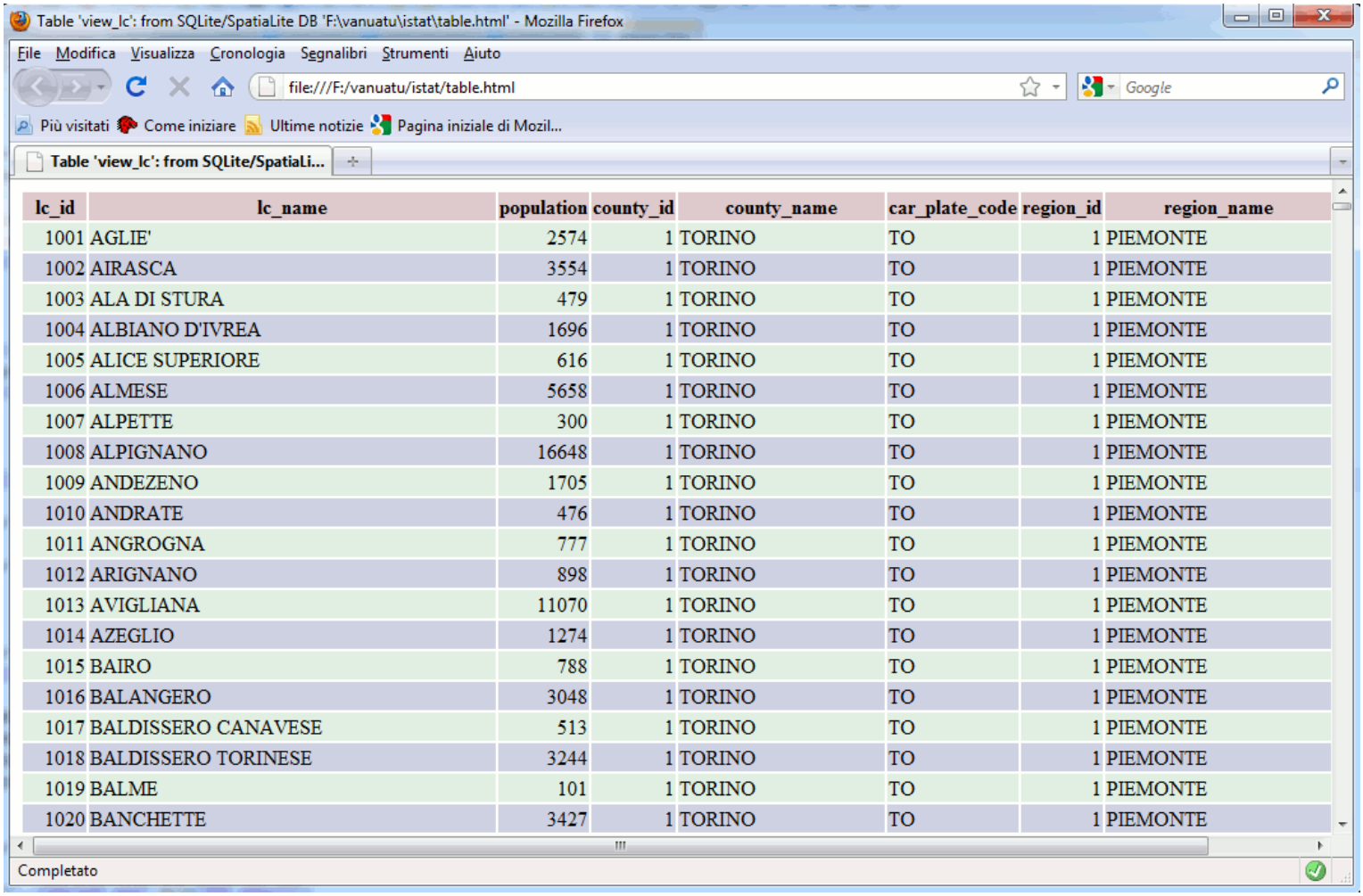
Other supported export formats

Using **spatialite_gui** you can also export your data as:

- HTML web pages
- PNG images (Geometries only)
- PDF documents (Geometries only)
- SVG vector graphics (Geometries only)

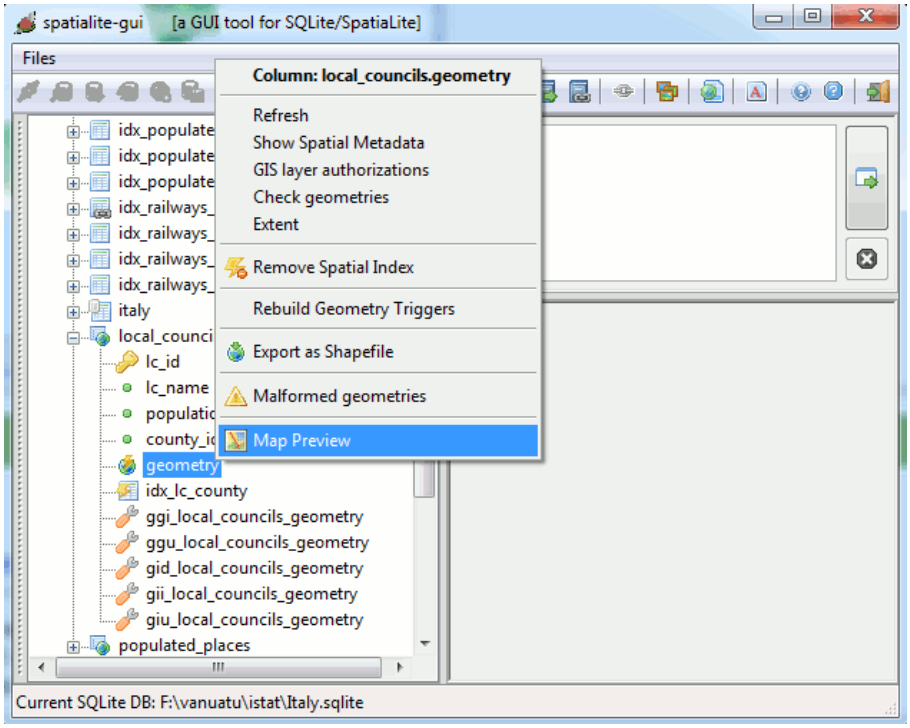


spatialite_gui: HTML export

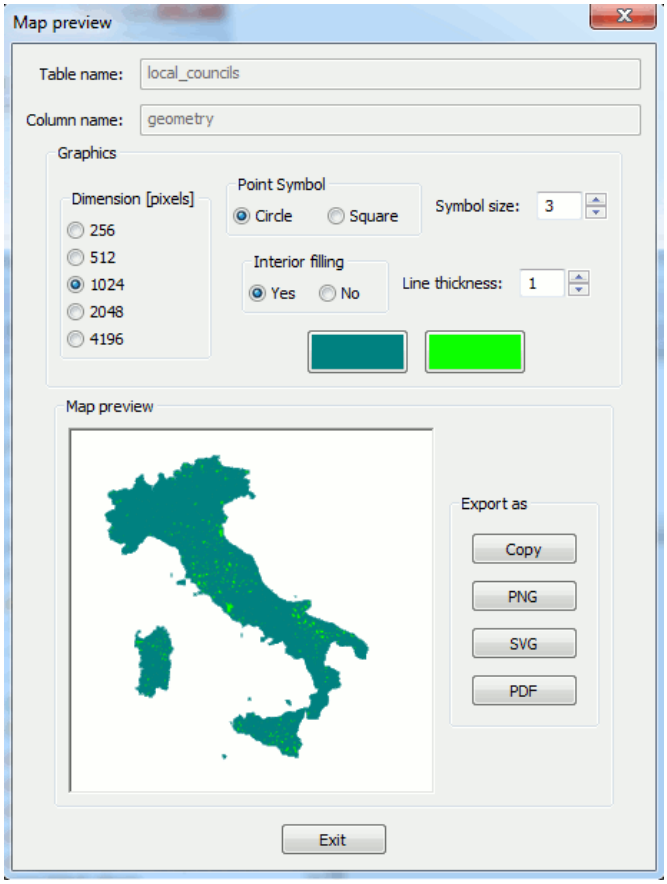


lc_id	lc_name	population	county_id	county_name	car_plate_code	region_id	region_name
1001	AGLIE'	2574	1	TORINO	TO	1	PIEMONTE
1002	AIRASCA	3554	1	TORINO	TO	1	PIEMONTE
1003	ALA DI STURA	479	1	TORINO	TO	1	PIEMONTE
1004	ALBIANO D'IVREA	1696	1	TORINO	TO	1	PIEMONTE
1005	ALICE SUPERIORE	616	1	TORINO	TO	1	PIEMONTE
1006	ALMESE	5658	1	TORINO	TO	1	PIEMONTE
1007	ALPETTE	300	1	TORINO	TO	1	PIEMONTE
1008	ALPIGNANO	16648	1	TORINO	TO	1	PIEMONTE
1009	ANDEZENO	1705	1	TORINO	TO	1	PIEMONTE
1010	ANDRATE	476	1	TORINO	TO	1	PIEMONTE
1011	ANGROGNA	777	1	TORINO	TO	1	PIEMONTE
1012	ARIGNANO	898	1	TORINO	TO	1	PIEMONTE
1013	AVIGLIANA	11070	1	TORINO	TO	1	PIEMONTE
1014	AZEGLIO	1274	1	TORINO	TO	1	PIEMONTE
1015	BAIRO	788	1	TORINO	TO	1	PIEMONTE
1016	BALANGERO	3048	1	TORINO	TO	1	PIEMONTE
1017	BALDISSERO CANAVESE	513	1	TORINO	TO	1	PIEMONTE
1018	BALDISSERO TORINESE	3244	1	TORINO	TO	1	PIEMONTE
1019	BALME	101	1	TORINO	TO	1	PIEMONTE
1020	BANCHETTE	3427	1	TORINO	TO	1	PIEMONTE

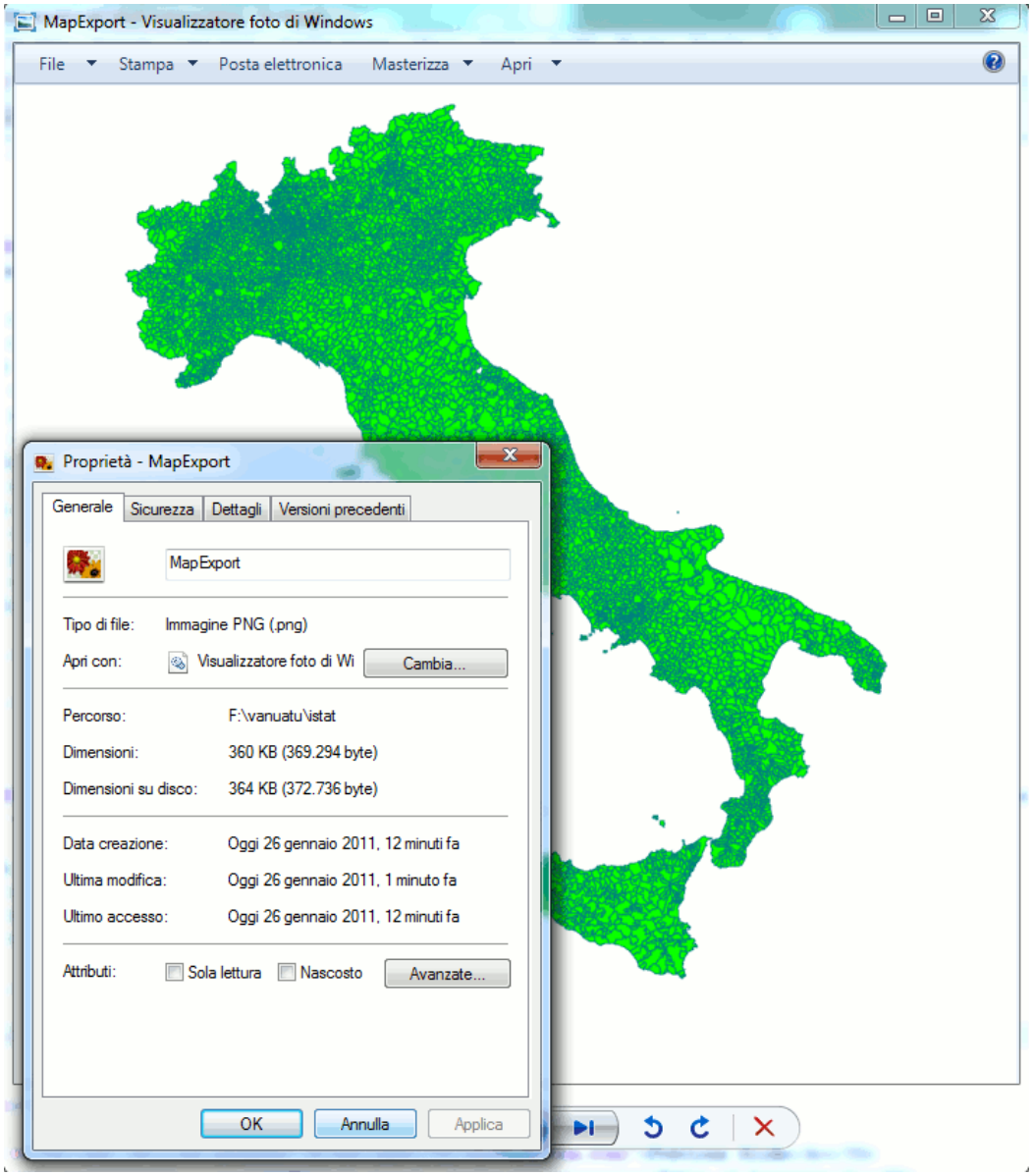
HTML export sample



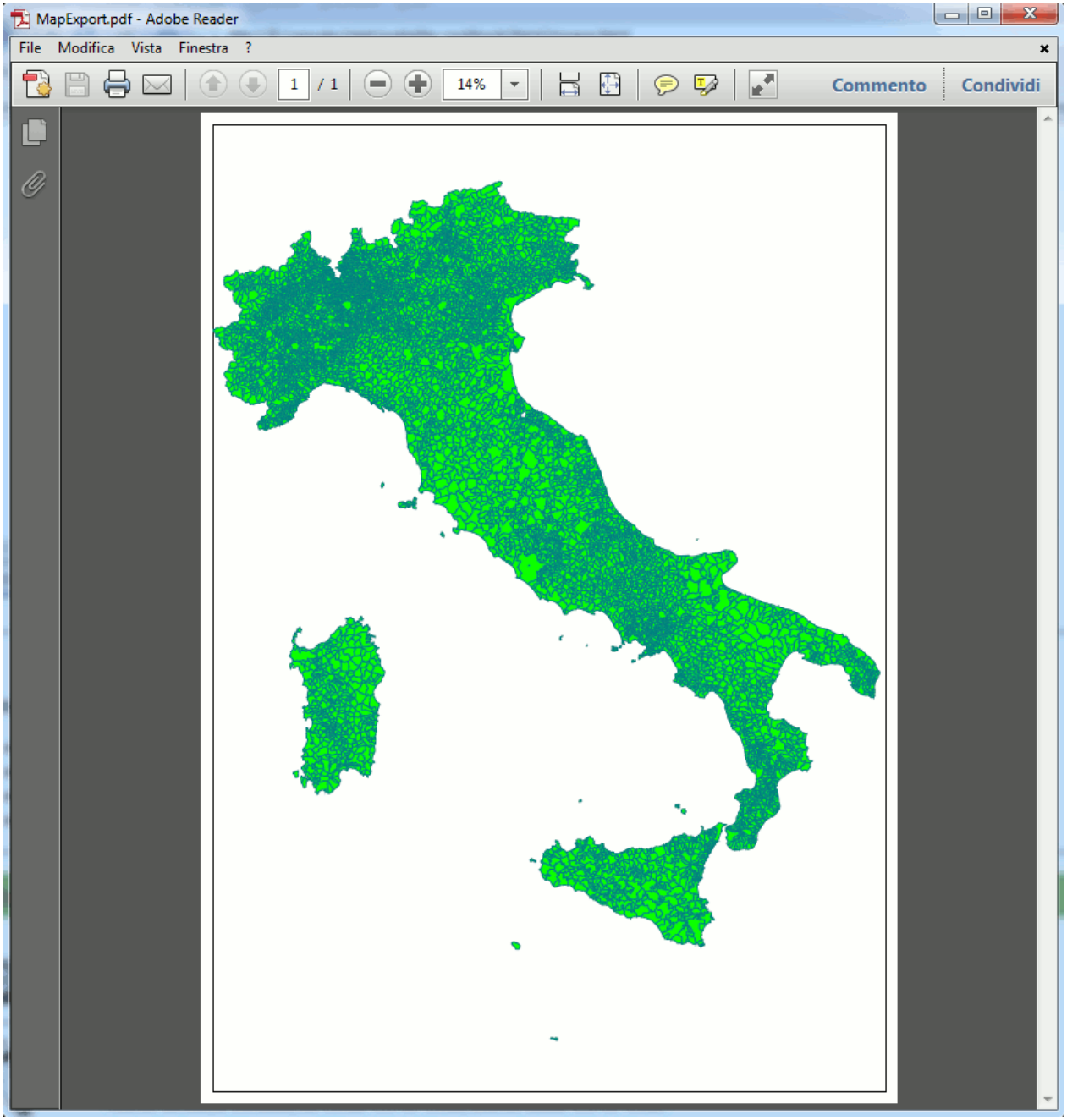
`spatialite_gui`: PNG / PDF / SVG export (1)



`spatialite_gui`: PNG / PDF / SVG export (2)



PNG export sample



PDF export sample

SVG export sample

Language bindings: [C/C++, Java, Python, PHP ...]

2011 January 28

Both SQLite and SpatiaLite are elementary simple are really lightweight.
So both them are obvious candidates when you are a software developer, and your application absolutely requires a robust and affordable Spatial DBMS support, but you are attempting to keep anything as simple as possible, possibly avoiding at all any unnecessary complexity.

The best fit development language supporting SQLite and SpatiaLite is obviously **C [**or its *ugly duckling* son, the **C++**]
(*after all, both SQLite and SpatiaLite are completely written in C language*).
Using **C/C++** you can directly access the wonderful **API**s of both SQLite and SpatiaLite, so to get a full and unconstrained access to any supported feature at the lowest possible level.
And that's not all: using **C/C++** you can eventually adopt **static linkage**, so to embed directly within the executable itself (*and with an incredibly small footprint*) a fully self-contained DBMS engine.
And such an approach surely gets rid of any installation related headache.

Anyway you are not at all compelled to necessarily use **C/C++**
SQLite and SpatiaLite are supported as well by many other languages such as **Java**, **Python**, **PHP** (*and probably many others*).

The basic approach

Any language supporting any generic **SQLite driver** *aka* **connector** can fully support SpatiaLite.
SQLite supports dynamic extension loading; and SpatiaLite simply is such an extension.

```
SELECT load_extension('path_to_extension_library');
```

Executing the above SQL statement will load any SQLite's extension: this obviously including SpatiaLite.
Anyway, too much often this is true only *in theory*, but *reality* is completely different from this.

Let us quickly examine the main issues actually bringing this simple approach to a complete failure:

- SQLite is highly configurable; it supports lots and lots of build-time options.
  You can completely disable at all the dynamic extension load mechanism, if you think this one could be a safe option.
  Sadly, for many long years this one has been the favorite choice for the vast majority of system packagers.
- SQLite is growing very quickly: usually a major update is released every few months.
  But many system packagers still continue distributing incredibly obsolete SQLite's versions.
  And quite obviously such obsolete versions cannot adequately support SpatiaLite.
- Once you are caught in this painful situation (disabled extensions / obsolete SQLite) you cannot do absolutely nothing.
  You simply have to give up forgetting SpatiaLite: at least for now.
- Anyway, if this is your actual case, doesn't lose heart: things evolves, and usually tends to evolve in the right direction.

Just two years ago very few languages supported SpatiaLite.
Today this isn't any longer true for the most widely used languages (*at least, using recently released versions*).

You can usefully read the appropriate section corresponding to your beloved language, so to get started in the quickest time.
Each section contains a *complete sample program*, and contains as well any related *system configuration* hint, *compiler/linker settings* and so on:

- C / C++
- Java / JDBC
- Python
- PHP

Language bindings: C/C++

2011 January 28

C/C++ sample program

spatialite_sample.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <float.h>

#ifdef SPATIALITE_AMALGAMATION
    #include <spatialite/sqlite3.h>
#else
    #include <sqlite3.h>
#endif


#ifndef SPATIALITE_EXTENSION
    #include <spatialite.h>
#endif

int
main (void)
{
    sqlite3 *db_handle;
    sqlite3_stmt *stmt;
    int ret;
    char *err_msg = NULL;
    char sql[2048];
    char sql2[1024];
    int i;
    char **results;
    int rows;
    int columns;
    int cnt;
    const char *type;
    int srid;
    char name[1024];
    char geom[2048];

#ifndef SPATIALITE_EXTENSION
    /*
     * initializing SpatiaLite-Amalgamation
     *
     * any C/C++ source requires this to be performed
     * for each connection before invoking the first
     * SQLite/SpatiaLite call
     */
    spatialite_init (0);
    fprintf(stderr, "\n\n******* hard-linked libspatialite ********\n\n");
#endif

/* creating/connecting the test_db */
    ret =
    sqlite3_open_v2 ("test-db.sqlite", &db_handle,
         SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, NULL);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "cannot open 'test-db.sqlite': %s\n",
         sqlite3_errmsg (db_handle));
      sqlite3_close (db_handle);
      db_handle = NULL;
```

```
      return -1;
      }

#ifdef SPATIALITE_EXTENSION
    /*
     * loading SpatiaLite as an extension
     */
    sqlite3_enable_load_extension (db_handle, 1);
    strcpy (sql, "SELECT load_extension('libspatialite.so')");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "load_extension() error: %s\n", err_msg);
      sqlite3_free (err_msg);
      return 0;
      }
    fprintf(stderr, "\n\n**** SpatiaLite loaded as an extension ***\n\n");
#endif

/* reporting version infos */
#ifndef SPATIALITE_EXTENSION
/*
* please note well:
* this process is physically linked to libspatialite
* so we can directly call any SpatiaLite's API function
*/
    fprintf (stderr, "SQLite version: %s\n", sqlite3_libversion());
    fprintf (stderr, "SpatiaLite version: %s\n", spatialite_version());
#else
/*
* please note well:
* this process isn't physically linked to libspatialite
* because we loaded the library as an extension
*
* so we aren't enabled to directly call any SpatiaLite's API functions
* we simply can access SpatiaLite indirectly via SQL statements
*/
    strcpy (sql, "SELECT sqlite_version()");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    if (rows < 1)
      {
      fprintf (stderr,
         "Unexpected error: sqlite_version() not found ??????\n");
      goto stop;
      }
    else
      {
      for (i = 1; i <= rows; i++)
       {
        fprintf (stderr, "SQLite version: %s\n",
         results[(i * columns) + 0]);
       }
      }
    sqlite3_free_table (results);
    strcpy (sql, "SELECT spatialite_version()");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    if (rows < 1)
      {
      fprintf (stderr,
         "Unexpected error: spatialite_version() not found ??????\n");
      goto stop;
      }
    else
      {
      for (i = 1; i <= rows; i++)
       {
```

```
        fprintf (stderr, "SpatiaLite version: %s\n",
         results[(i * columns) + 0]);
       }
      }
    sqlite3_free_table (results);
#endif /* SpatiaLite as an extension */

/* initializing SpatiaLite's metadata tables */
    strcpy (sql, "SELECT InitSpatialMetadata()");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "InitSpatialMetadata() error: %s\n", err_msg);
      sqlite3_free (err_msg);
      return 0;
      }

/* creating a POINT table */
    strcpy (sql, "CREATE TABLE test_pt (");
    strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
    strcat (sql, "name TEXT NOT NULL)");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
/* creating a POINT Geometry column */
    strcpy (sql, "SELECT AddGeometryColumn('test_pt', ");
    strcat (sql, "'geom', 4326, 'POINT', 'XY')");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }

/* creating a LINESTRING table */
    strcpy (sql, "CREATE TABLE test_ln (");
    strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
    strcat (sql, "name TEXT NOT NULL)");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
/* creating a LINESTRING Geometry column */
    strcpy (sql, "SELECT AddGeometryColumn('test_ln', ");
    strcat (sql, "'geom', 4326, 'LINESTRING', 'XY')");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }

/* creating a POLYGON table */
    strcpy (sql, "CREATE TABLE test_pg (");
    strcat (sql, "id INTEGER NOT NULL PRIMARY KEY,");
    strcat (sql, "name TEXT NOT NULL)");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
/* creating a POLYGON Geometry column */
    strcpy (sql, "SELECT AddGeometryColumn('test_pg', ");
    strcat (sql, "'geom', 4326, 'POLYGON', 'XY')");
    ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
```

```
      goto stop;
      }

/*
* inserting some POINTs
* please note well: SQLite is ACID and Transactional
* so (to get best performance) the whole insert cycle
* will be handled as a single TRANSACTION
*/
    ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    for (i = 0; i < 100000; i++)
      {
      /* for POINTs we'll use full text sql statements */
      strcpy (sql, "INSERT INTO test_pt (id, name, geom) VALUES (");
      sprintf (sql2, "%d, 'test POINT #%d'", i + 1, i + 1);
      strcat (sql, sql2);
      sprintf (sql2, ", GeomFromText('POINT(%1.6f %1.6f)'", i / 1000.0,
         i / 1000.0);
      strcat (sql, sql2);
      strcat (sql, ", 4326))");
      ret = sqlite3_exec (db_handle, sql, NULL, NULL, &err_msg);
      if (ret != SQLITE_OK)
       {
        fprintf (stderr, "Error: %s\n", err_msg);
        sqlite3_free (err_msg);
        goto stop;
       }
      }
    ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }

/* checking POINTs */
    strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
    strcat (sql, "ST_Srid(geom) FROM test_pt");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    if (rows < 1)
      {
      fprintf (stderr, "Unexpected error: ZERO POINTs found ??????\n");
      goto stop;
      }
    else
      {
      for (i = 1; i <= rows; i++)
       {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
         cnt, type, srid);
       }
      }
    sqlite3_free_table (results);

/*
* inserting some LINESTRINGs
* this time we'll use a Prepared Statement
*/
    strcpy (sql, "INSERT INTO test_ln (id, name, geom) ");
    strcat (sql, "VALUES (?, ?, GeomFromText(?, 4326))");
    ret = sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
    if (ret != SQLITE_OK)
      {
```

```
      fprintf (stderr, "SQL error: %s\n%s\n", sql,
         sqlite3_errmsg (db_handle));
      goto stop;
      }
    ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    for (i = 0; i < 100000; i++)
      {
      /* setting up values / binding */
      sprintf (name, "test LINESTRING #%d", i + 1);
      strcpy (geom, "LINESTRING(");
      if ((i % 2) == 1)
       {
        /* odd row: five points */
        strcat (geom, "-180.0 -90.0, ");
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
         -10.0 - (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
         10.0 + (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
         10.0 + (i / 1000.0));
        strcat (geom, sql2);
        strcat (geom, "180.0 90.0");
       }
      else
       {
        /* even row: two points */
        sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
         -10.0 - (i / 1000.0));
        strcat (geom, sql2);
        sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
         10.0 + (i / 1000.0));
        strcat (geom, sql2);
       }
      strcat (geom, ")");
      sqlite3_reset (stmt);
      sqlite3_clear_bindings (stmt);
      sqlite3_bind_int (stmt, 1, i + 1);
      sqlite3_bind_text (stmt, 2, name, strlen (name), SQLITE_STATIC);
      sqlite3_bind_text (stmt, 3, geom, strlen (geom), SQLITE_STATIC);
      /* performing INSERT INTO */
      ret = sqlite3_step (stmt);
      if (ret == SQLITE_DONE || ret == SQLITE_ROW)
       continue;
      fprintf (stderr, "sqlite3_step() error: [%s]\n",
         sqlite3_errmsg (db_handle));
      goto stop;
      }
    sqlite3_finalize (stmt);
    ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }

/* checking LINESTRINGs */
    strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
    strcat (sql, "ST_Srid(geom) FROM test_ln");
    ret =
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    if (rows < 1)
      {
      fprintf (stderr, "Unexpected error: ZERO LINESTRINGs found ??????\n");
      goto stop;
      }
```

```
    else
      {
      for (i = 1; i <= rows; i++)
       {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
         cnt, type, srid);
       }
      }
    sqlite3_free_table (results);

/*
* inserting some POLYGONs
* this time too we'll use a Prepared Statement
*/
    strcpy (sql, "INSERT INTO test_pg (id, name, geom) ");
    strcat (sql, "VALUES (?, ?, GeomFromText(?, 4326))");
    ret = sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "SQL error: %s\n%s\n", sql,
         sqlite3_errmsg (db_handle));
      goto stop;
      }
    ret = sqlite3_exec (db_handle, "BEGIN", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    for (i = 0; i < 100000; i++)
      {
      /* setting up values / binding */
      sprintf (name, "test POLYGON #%d", i + 1);
      strcpy (geom, "POLYGON((");
      sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
         -10.0 - (i / 1000.0));
      strcat (geom, sql2);
      sprintf (sql2, "%1.6f %1.6f, ", 10.0 - (i / 1000.0),
         -10.0 - (i / 1000.0));
      strcat (geom, sql2);
      sprintf (sql2, "%1.6f %1.6f, ", 10.0 + (i / 1000.0),
         10.0 + (i / 1000.0));
      strcat (geom, sql2);
      sprintf (sql2, "%1.6f %1.6f, ", -10.0 - (i / 1000.0),
         10.0 - (i / 1000.0));
      strcat (geom, sql2);
      sprintf (sql2, "%1.6f %1.6f", -10.0 - (i / 1000.0),
         -10.0 - (i / 1000.0));
      strcat (geom, sql2);
      strcat (geom, "))");
      sqlite3_reset (stmt);
      sqlite3_clear_bindings (stmt);
      sqlite3_bind_int (stmt, 1, i + 1);
      sqlite3_bind_text (stmt, 2, name, strlen (name), SQLITE_STATIC);
      sqlite3_bind_text (stmt, 3, geom, strlen (geom), SQLITE_STATIC);
      /* performing INSERT INTO */
      ret = sqlite3_step (stmt);
      if (ret == SQLITE_DONE || ret == SQLITE_ROW)
       continue;
      fprintf (stderr, "sqlite3_step() error: [%s]\n",
         sqlite3_errmsg (db_handle));
      goto stop;
      }
    sqlite3_finalize (stmt);
    ret = sqlite3_exec (db_handle, "COMMIT", NULL, NULL, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }

/* checking POLYGONs */
    strcpy (sql, "SELECT DISTINCT Count(*), ST_GeometryType(geom), ");
    strcat (sql, "ST_Srid(geom) FROM test_pg");
    ret =
```

```
    sqlite3_get_table (db_handle, sql, &results, &rows, &columns, &err_msg);
    if (ret != SQLITE_OK)
      {
      fprintf (stderr, "Error: %s\n", err_msg);
      sqlite3_free (err_msg);
      goto stop;
      }
    if (rows < 1)
      {
      fprintf (stderr, "Unexpected error: ZERO POLYGONs found ??????\n");
      goto stop;
      }
    else
      {
      for (i = 1; i <= rows; i++)
       {
        cnt = atoi (results[(i * columns) + 0]);
        type = results[(i * columns) + 1];
        srid = atoi (results[(i * columns) + 2]);
        fprintf (stderr, "Inserted %d entities of type %s SRID=%d\n",
         cnt, type, srid);
       }
      }
    sqlite3_free_table (results);

/* closing the DB connection */
  stop:
    sqlite3_close (db_handle);
    return 0;
}
```

Compiling and linking

SpatiaLite comes in different flavors:

- **`libspatialite`** is the standard library intended to be loaded as an extension.
- **`libspatialite-amalgamation`** is a self-standing complete ***SQL-engine*** supporting an *internal private copy* of SQLite.

If your principal interest is developing easy-to-be-deployed, stand-alone C/C++ applications, then using the ***statically linked*** **`libspatialite-amalgamation`** will surely be a desirable option.
Anyway nothing prevents you from using a different layout based on ***dynamically linked*** shared libraries.
And a third option is supported as well: you can completely avoid using SQLite and SpatiaLite libraries.
Both them simply are *one single monolithic source file*: so you can directly compile any required source in a single pass (*this will obviously generate a* ***statically linked*** *executable*).

Get a quick glance to the **`spatialite_sample.c`** source code; you easily notice that two ***conditional macros*** are extensively used:

- **`SPATIALITE_AMALGAMATION`** is used to determine if we are using separate **`libsqlite`** and **`libspatialite`**, or if we are using the all-in-one **`libspatialite-amalgamation`**
  - as you can notice the only difference affects inclusion of header files:
    when using **amalgamation** you cannot use **`#include <sqlite3.h>`** (*because this one is the standard* ***system sqlite*** *header file*).
  - you are required using **`#include <spatialite/sqlite3.h>`** (*a purposely modified version supporting* ***amalgamation***).
- **`SPATIALITE_EXTENSION`** is used to determine if we are using an *hard-linked* **`libspatialite`** or if we intend loading this library as an extension:
  - **please note:** loading **`libspatialite`** as an extension excludes using the **amalgamation**, because **`libsqlite`** is already linked to the executable in this case.

You can now build **spatialite_sample.c** in many different flavors, simply defining these two macros as

appropriate, in the most flexible way.
The following practical examples are based on the *standard GNU* **gcc** compiler on Linux.

Loading SpatiaLite as an extension

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_EXTENSION \
  spatialite_sample.c -o spatialite_sample -lsqlite3
```

Some interesting points to be noted:

- you must not explicitly link **libspatialite**;
  this is because we'll *dynamically load* this library only at run-time.
- the standard *system* **libsqlite** will be used in this case.
- please note #1: this approach strongly simplifies compile and link phases, but simply postpones the problem until run-time.
  And obviously a main failure will surely arise if (*for any reason*) the extension library cannot be actually loaded.
- please note #2: adopting this approach your executable cannot directly access any SpatiaLite's own function, because no **libspatialite** was hard-linked;
  and consequently any *export symbol* defined into this library is actually *invisible* to the executable code.
  Anyway, you can safely invoke any function exposed via SQL language, once **libsqlite** has successfully loaded the extension library.

This basic approach is exactely the same supported by any other language binding (Java, Python, PHP ...).
But using C/C++ you get get more flexible configuration options: please, see the following examples.

Dynamically linking libspatialite + libsqlite

```
gcc -Wall -Wextra -Wunused -I/usr/local/include \
  spatialite_sample.c -o spatialite_sample
  -L/usr/local/lib -lspatialite -lsqlite3
```

Some interesting points to be noted:

- this time we'll *hard-link* **libspatialite** to our executable.
- we'll **not** use **amalgamation**, so neither **-DSPATIALITE_AMALGAMATION** nor **-DSPATIALITE_EXTENSION** have to be declared this time.
- usually **libspatialite** is installed on the **/usr/local** directory:
  so we'll set **-I** and **-L** options in order to extend the searching rules for header files and libraries.
- please note: this time we'll link both **libspatialite** and **libsqlite**

Dynamically linking libspatialite-amalgamation

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
  -I/usr/local/include spatialite_sample.c \
```

```
  -o spatialite_sample \
  -L/usr/local/lib -lspatialite
```

Some interesting points to be noted:

- this time we'll use **libspatialite-amalgamation**
- and accordingly to this, we'll define **-DSPATIALITE_AMALGAMATION**
- as in the previous example we'll set **-I** and **-L** options in order to extend the searching rules for header files and libraries.
- please note: this time will simply link **libspatialite**;
  there is absolutely no need to link **libsqlite** as well, because we'll use the *private internal copy* directly included within **libspatialite-amalgamation**

Statically linking libspatialite + libsqlite

```
gcc -Wall -Wextra -Wunused -I/usr/local/include \
  spatialite_sample.c -o spatialite_sample \
  /usr/local/lib/libspatialite.a \
  /usr/lib/libsqlite.a \
  /usr/lib/libgeos_c.a \
  /usr/lib/libgeos.a \
  /usr/lib/libproj.a \
  -lstdc++ -ldl -lpthread -lm
```

Some interesting points to be noted:

- adopting a *statically linked* strategy we are consequently required to resolve any symbol at link time.
  So we have to explicitly refer several further libraries:
  - **libproj**, **libgeos_c** and **libgeos** are always required.
  - **-lstdc++** is required as well, because **libgeos** actually is a C++ library, thus requiring the C++ run-time support.
  - on Linux **-ldl** and **-lpthread** are strictly required.
  - on some Linux declaring **-lm** may be required as well (*and is always harmless, even when not strictly required*).

Statically linking libspatialite-amalgamation

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
  -I/usr/local/include spatialite_sample.c \
  -o spatialite_sample \
  /usr/local/lib/libspatialite.a \
  /usr/lib/libgeos_c.a \
  /usr/lib/libgeos.a \
  /usr/lib/libproj.a \
  -lstdc++ -ldl -lpthread -lm
```

Exactly the same as above; simply this time **libsqlite** is no longer required, because this is directly supported by the *private internal copy* included within the **amalgamation** itself.

Please note: different platforms, different system libraries, different file system layouts.
The **gcc** compiler is available on quite every platform (*this including Microsoft Windows, of course*).
Unhappily, each platform has its own specific idiosyncrasies.

MacOsX:

```
```

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
  -I/usr/local/include -I/opt/local/include
  spatialite_sample.c -o spatialite_sample \
  /usr/local/lib/libspatialite.a \
  /opt/local/lib/libgeos_c.a \
  /opt/local/lib/libgeos.a \
  /opt/local/lib/libproj.a \
  /opt/local/lib/libiconv.a \
  /opt/local/lib/libcharset.a \
  -lstdc++ -ldl -lpthread -lm
```

MacOsX basically is an Unix derivative (*most precisely a FreeBSD derivative*).
The wonderful MacPorts fully supports *standard open source* libraries (in our case **`libgeos`**, **`libproj`** and **`libiconv`**):

- **MacPorts** header files will be installed on **`/opt/local/include`**
- **MacPorts** libraries will be installed on **`/opt/local/lib`**
- Both **`libiconv`** and **`libcharset`** are directly integrated within the C run-time on Linux.
  But on MacOsX you are required to explicitly link these libraries.

Windows [MinGW + MSYS]:

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
  -I/usr/local/include spatialite_sample.c \
  -o spatialite_sample.exe \
  /usr/local/lib/libspatialite.a \
  /usr/lib/libgeos_c.a \
  /usr/lib/libgeos.a \
  /usr/lib/libproj.a \
  /usr/local/lib/libiconv.a \
  -lstdc++ -lm
```

Windows and Unix are strongly different:

- as we have already seen for MacOsX explicitly linking **`libiconv`** is required for Windows as well (but not **`libcharset`**).
- **`-ldl`** and **`-lpthread`** aren't required at all
  (*simply because Windows **dynamic loader** and **multithreading** are completely different from the corresponding Unix implementations, and are directly supported by the **WIN32** run-time*).

Simplest approach: no libs at all ...

```
gcc -Wall -Wextra -Wunused -DSPATIALITE_AMALGAMATION \
  -DOMIT_GEOS=1 -DOMIT_PROJ=1 -DOMIT_ICONV=1 \
  -DOMIT_GEOCALLBACKS=1 \
  -I./headers -I/usr/local/include \
  spatialite.c sqlite3.c spatialite_sample.c \
  -o spatialite_sample.exe
```

And finally a *third way* exists, one allowing to directly embed a complete Spatial SQL engine in the simplest and absolutely painless way.
[*I suppose following this latest approach you can be actually able to get a minimal SpatiaLite support even on **PDA** or **smart-phones***]
Just few explanations:

- you must create a new directory.
- copy the **`spatialite_sample.c`** source into this directory.
- then copy into the same directory the **`sqlite3.c`** and **`spatialite.c`** sources
  (*they are included in any source distribution of **`libspatialite-amalgamation`***).
- and finally you must copy the whole **`headers`** directory from **`libspatialite-amalgmation`**sources.

All right, you are now ready to build all-together your *statically linked* executable.
Please note, absolutely no external libraries are required:

- this build is based on **`libspatialite-amalgamation`**, so we'll define **`-DSPATIALITE_AMALGAMATION`**
- by defining **`-DOMIT_GEOS=1`** (*and so on*) we'll completely disable GEOS, PROJ.4 and ICONV support.
- but this will avoid at all linking any further library.
- yes, that's true, now SpatiaLite is severely gilded:
  but the **main-core** of Spatial SQL is preserved untouched.
  And that's more than enough for many and many practical purposes.

Language bindings: Java / JDBC

2011 January 28

Test environment

Linux Debian:
just to be sure to test the up-to-date state-of-the-art I've actually used **Debian Squeeze** (32 bit).
This way I'm sure that all required packages are reasonably using the most recent version.

Java:
I was really curious about **OpenJava** (I had never used it before), so I decided not to use Sun Java.
After the first initial troubles I discovered I had to install the following packages:

- **openjdk-6-jre** (Java JDK)
- **gcj** (Java compiler)

SQLite's JDBC connector:
I used the one from http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC
I've actually downloaded the latest supported version:
http://www.xerial.org/maven/repository/artifact/org/xerial/sqlite-jdbc/3.7.2/sqlite-jdbc-3.7.2.jar
Btw this one isn't the latest version supported by SQLite, because v.3.7.3 was released on 2010 October 8, and v.3.7.4 is available since 2010 December 8: so the Xerial support seems to be slightly outdated. Not a big issue, anyway.

My first test

Just to check if my environment was a valid one at first I simply compiled and then ran the standard Xerial demo.
You'll easily find the code on their main HTML page about JDBC.
I simply copied the Java code from the HTML page into a text file named **Sample.java**

Important notice: following Xerial instructions I simply copied the **sqlite-jdbc-3.7.2.jar** file directly under the same directory where I placed the **Sample.java** source, so to avoid any CLASSPATH related headache.

```
$ javac Sample.java
$ java -classpath ".:sqlite-jdbc.3.7.2.jar" Sample
```

All right: everything worked as expected. Anyway this first test simply stressed basic SQLite capabilities.
I had now going further on, attempting to test if the Xerial JDBC connector could actually support SpatiaLite.

First SpatiaLite test (failure)

The most recent **libspatialite-2.4.0-RC4** was already installed on my test platform (actually this is one of the Linux workhorses I currently use for development and testing).
I had built this package by myself, so the corresponding shared library was **/usr/local/lib/libspatialite.so**

The first obvious thing to be done was loading the SpatiaLite's shared library, so to enable the JDBC connector supporting SpatiaLite.
So I simply added the following line to the souce code (immediately after establishing the connection).

```
stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so');
```

Too much simplistic: this way I've got a discouraging error message: **NOT AUTHORIZED**.
After a little while I've found an useful suggestion browsing the Web: a preliminary step is absolutely required in order to enable extension dynamic loading.

```
import org.sqlite.SQLiteConfig;
...
SQLiteConfig config = new SQLiteConfig;
config.enableLoadExtension(true);
Connection conn = DriverManager.getConnection("path", config.toProperties);
Statement stmt = conn.createStatement();
stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so')");
```

All right; now the SpatiaLite's shared library was successfully loaded.
And this one was the unique misadventure I experienced during my JDBC testing: once I was able resolving this issue then anything ran absolutely smooth and without any further accident.
Except for one JDBC oddity I noticed: but I'll account for this at the end of the story.

Java sample program

SpatialiteSample.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;
import org.sqlite.SQLiteConfig;

public class SpatialiteSample
{
  public static void main(String[] args) throws ClassNotFoundException
  {
    // load the sqlite-JDBC driver using the current class loader
    Class.forName("org.sqlite.JDBC");

    Connection conn = null;
    try
    {
      // enabling dynamic extension loading
      // absolutely required by SpatiaLite
      SQLiteConfig config = new SQLiteConfig();
      config.enableLoadExtension(true);

      // create a database connection
      conn = DriverManager.getConnection("jdbc:sqlite:spatialite.sample",
      config.toProperties());
      Statement stmt = conn.createStatement();
      stmt.setQueryTimeout(30); // set timeout to 30 sec.

      // loading SpatiaLite
      stmt.execute("SELECT load_extension('/usr/local/lib/libspatialite.so')");

      // enabling Spatial Metadata
      // using v.2.4.0 this automatically initializes SPATIAL_REF_SYS and GEOMETRY_COLUMNS
      String sql = "SELECT InitSpatialMetadata()";
      stmt.execute(sql);

      // creating a POINT table
      sql = "CREATE TABLE test_pt (";
      sql += "id INTEGER NOT NULL PRIMARY KEY,";
      sql += "name TEXT NOT NULL)";
      stmt.execute(sql);
      // creating a POINT Geometry column
      sql = "SELECT AddGeometryColumn('test_pt', ";
      sql += "'geom', 4326, 'POINT', 'XY')";
      stmt.execute(sql);

      // creating a LINESTRING table
      sql = "CREATE TABLE test_ln (";
      sql += "id INTEGER NOT NULL PRIMARY KEY,";
      sql += "name TEXT NOT NULL)";
      stmt.execute(sql);
      // creating a LINESTRING Geometry column
      sql = "SELECT AddGeometryColumn('test_ln', ";
      sql += "'geom', 4326, 'LINESTRING', 'XY')";
      stmt.execute(sql);

      // creating a POLYGON table
      sql = "CREATE TABLE test_pg (";
      sql += "id INTEGER NOT NULL PRIMARY KEY,";
      sql += "name TEXT NOT NULL)";
      stmt.execute(sql);
      // creating a POLYGON Geometry column
      sql = "SELECT AddGeometryColumn('test_pg', ";
      sql += "'geom', 4326, 'POLYGON', 'XY')";
```

```
stmt.execute(sql);

// inserting some POINTs
// please note well: SQLite is ACID and Transactional,
// so (to get best performance) the whole insert cycle
// will be handled as a single TRANSACTION
conn.setAutoCommit(false);
int i;
for (i = 0; i < 100000; i++)
{
  // for POINTs we'll use full text sql statements
  sql = "INSERT INTO test_pt (id, name, geom) VALUES (";
  sql += i + 1;
  sql += ", 'test POINT #";
  sql += i + 1;
  sql += "', GeomFromText('POINT(";
  sql += i / 1000.0;
  sql += " ";
  sql += i / 1000.0;
  sql += ")', 4326))";
  stmt.executeUpdate(sql);
}
conn.commit();

// checking POINTs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
sql += "ST_Srid(geom) FROM test_pt";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next())
{
  // read the result set
  String msg = "> Inserted ";
  msg += rs.getInt(1);
  msg += " entities of type ";
  msg += rs.getString(2);
  msg += " SRID=";
  msg += rs.getInt(3);
  System.out.println(msg);
}

// inserting some LINESTRINGs
// this time we'll use a Prepared Statement
sql = "INSERT INTO test_ln (id, name, geom) ";
sql += "VALUES (?, ?, GeomFromText(?, 4326))";
PreparedStatement ins_stmt = conn.prepareStatement(sql);
conn.setAutoCommit(false);
for (i = 0; i < 100000; i++)
{
  // setting up values / binding
  String name = "test LINESTRING #";
  name += i + 1;
  String geom = "LINESTRING (";
  if ((i%2) == 1)
  {
    // odd row: five points
    geom += "-180.0 -90.0, ";
    geom += -10.0 - (i / 1000.0);
    geom += " ";
    geom += -10.0 - (i / 1000.0);
    geom += ", ";
    geom += -10.0 - (i / 1000.0);
    geom += " ";
    geom += 10.0 + (i / 1000.0);
    geom += ", ";
    geom += 10.0 + (i / 1000.0);
    geom += " ";
    geom += 10.0 + (i / 1000.0);
    geom += ", 180.0 90.0";
  }
  else
  {
    // even row: two points
    geom += -10.0 - (i / 1000.0);
    geom += " ";
    geom += -10.0 - (i / 1000.0);
    geom += ", ";
    geom += 10.0 + (i / 1000.0);
    geom += " ";
    geom += 10.0 + (i / 1000.0);
  }
  geom += ")";
  ins_stmt.setInt(1, i+1);
  ins_stmt.setString(2, name);
  ins_stmt.setString(3, geom);
  ins_stmt.executeUpdate();
}
conn.commit();

// checking LINESTRINGs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
sql += "ST_Srid(geom) FROM test_ln";
rs = stmt.executeQuery(sql);
while(rs.next())
{
  // read the result set
  String msg = "> Inserted ";
  msg += rs.getInt(1);
  msg += " entities of type ";
  msg += rs.getString(2);
  msg += " SRID=";
  msg += rs.getInt(3);
  System.out.println(msg);
}

// inserting some POLYGONs
// this time too we'll use a Prepared Statement
sql = "INSERT INTO test_pg (id, name, geom) ";
sql += "VALUES (?, ?, GeomFromText(?, 4326))";
ins_stmt = conn.prepareStatement(sql);
conn.setAutoCommit(false);
for (i = 0; i < 100000; i++)
{
  // setting up values / binding
  String name = "test POLYGON #";
  name += i + 1;
```

```
        ins_stmt.setInt(1, i+1);
        ins_stmt.setString(2, name);
        String geom = "POLYGON((";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += ", ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += ", ";
        geom += 10.0 + (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += ", ";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += 10.0 + (i / 1000.0);
        geom += ", ";
        geom += -10.0 - (i / 1000.0);
        geom += " ";
        geom += -10.0 - (i / 1000.0);
        geom += "))";
        ins_stmt.setInt(1, i+1);
        ins_stmt.setString(2, name);
        ins_stmt.setString(3, geom);
        ins_stmt.executeUpdate();
      }
      conn.commit();

      // checking POLYGONs
      sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
      sql += "ST_Srid(geom) FROM test_pg";
      rs = stmt.executeQuery(sql);
      while(rs.next())
      {
        // read the result set
        String msg = "> Inserted ";
        msg += rs.getInt(1);
        msg += " entities of type ";
        msg += rs.getString(2);
        msg += " SRID=";
        msg += rs.getInt(3);
        System.out.println(msg);
      }
    }
    catch(SQLException e)
    {
      // if the error message is "out of memory",
      // it probably means no database file is found
      System.err.println(e.getMessage());
    }
    finally
    {
      try
      {
        if(conn != null)
          conn.close();
      }
      catch(SQLException e)
      {
        // connection close failed.
        System.err.println(e);
      }
    }
  }
}
```

```
$ javac -classpath ".:sqlite-jdbc.3.7.2.jar" SpatialiteSample.java
$ java -classpath ".:sqlite-jdbc.3.7.2.jar" SpatialiteSample
```

Telling the full story is boring and very few interesting: you can study and test the sample code by yourself, and that's absolutely all.

Conclusions

1. the Xerial JDBC connector seems to be perfectly aimed to support SpatiaLite
2. performance is really good: more or less, exactly the same you can get using a C-based process.
   But that's not too much surprising after all: using the Xerial JDBC connector we are actually using binary C libraries, and Java merely acts like a wrapping shell.

Caveat

Although the Xerial JDBC connector seems to be really good, I noticed several potential flaws.
Very shortly said, your SQL statements have to be absolutely clean and well tested: because when the JDBC connector encounters some invalid SQL (*not at all an exceptional condition during the development life-cycle*),

it's most probable you'll then get a fatal JVM crash than a soft error exception.

I became quite crazy attempting to identify the cause for so frequent crashes during my tests: untill I finally realized that the problem simply was some stupid missing braket or quotation mark in complex SQL statements.
C can safely survive to all this without any damage, nicely reporting a soft and polite error message.
On the other side JDBC / JVM are unexorably unforgiving (and unstable) when they handle such trivial errors.

JDBC oddities

As I previously stated, I noticed a real JDBC oddity. It's now time to explain better this stupid issue.

C language snippet / SQLite API

```
strcpy(sql, "INSERT INTO xxx (id, geometry) VALUES (?, ");
strcat(sql, "GeomFromText('POINT(? ?, ? ?)', 4326))";
sqlite3_prepare_v2 (db_handle, sql, strlen (sql), &stmt, NULL);
sqlite3_bind_int (stmt, 0, 1);
sqlite3_bind_double (stmt, 1, 10.01);
sqlite3_bind_double (stmt, 2, 20.02);
sqlite3_bind_double (stmt, 3, 30.03);
sqlite3_bind_double (stmt, 4, 40.04);
sqlite3_step (stmt);
```

- we'll use a Prepared Statement to perform an INSERT INTO
- the first argument corresponds to the ID column
- any other argument corresponds to POINT coords
- all this runs absolutely smooth, because SQLite simply applies values substitution according to coded instructions.
  If you (as a developer) define some mismatching data-type, you'll then get some SQL error.
  This sound nice and fine: after all the developer has full authority (and responsibility ...)

Java language snippet / JDBC

```
sql = "INSERT INTO xxx (id, geometry) VALUES (?, ";
sql += "GeomFromText('POINT(? ?, ? ?)', 4326))";
stmt = conn.prepareStatement(sql);
stmt.setInt(1, 1);
stmt.setDouble(2, 10.01);
stmt.setDouble(3, 20.02);
stmt.setDouble(4, 30.03);
stmt.setDouble(5, 40.04);
stmt.executeUpdate();
```

- oops … this will fail on Java: we'll get a rather mysterious **array-out-of-bounds** fatal exception

Post-mortem: JDBC attempts to be smarter than you. While parsing the Prepared Statement JDBC discovers your dirty trick: the latest four args are enclosed within single quotes, so JDBC simply ignores them at all, because it intends the string literal as an *absolutely untouchable* entity.
You can check by yourself using **ParameterMetaData.getParameterCount()**; this prepared statement simply expects a single arg to be bounded.

Language bindings: Python

2011 January 28

Test environment

Linux Debian:
just to be sure to check an up-to-date state-of-the-art I've actually used **Debian Squeeze** (32 bit). So I'm actually sure that all required packages are reasonably using the most recent version.

Python:
python-2.6.6 was already installed on my testbed system, so I was immediately ready to start my test.

pyspatialite connector:
the connector source is available for download at: http://code.google.com/p/pyspatialite/
I've actually downloaded the latest supported version: http://pyspatialite.googlecode.com/files/pyspatialite-2.6.1.tar.gz
In order to build and install the **pyspatialite** connector I used the canonical Python scripts:

```
$ python setup.py build
... verbose output follows [suppressed] ...
$ su
# python setup.py install
```

Very important notice: I soon discovered that simply using the standard build script wasn't enough: this way I got an obsolete SpatiaLite **v.2.3.1** and that's not at all a good thing.

Caveat

The most recent QGIS versions (**1.6 / 1.7 trunk**) are actually using SpatiaLite **v.2.4.0** so any Python plugin using the previous v.2.3.1 can easily cause conflicts due to version incompatilibiles.
And this fully accounts for any issue noticed by QGIS users.

Patching setup.py

I quickly discovered that supporting the most recent SpatiaLite v.2.4.0-RC4 was actually a piece of cake: I simply had to change two lines in the **setup.py** script (**line 87** and followings).

```
def get_amalgamation():
    """Download the Spatialite amalgamation if it isn't there, already."""
    if os.path.exists(AMALGAMATION_ROOT):
        return
    os.mkdir(AMALGAMATION_ROOT)
    print "Downloading amalgation."

    # find out what's current amalgamation ZIP file
    download_page = urllib.urlopen("http://www.gaia-gis.it/spatialite-2.4.0-
4/sources.html").read()
```

```
    pattern = re.compile("(libspatialite-amalgamation.*?\.zip)")
    download_file = pattern.findall(download_page)[0]
    amalgamation_url = "http://www.gaia-gis.it/spatialite-2.4.0-4/"+ download_file
    zip_dir = string.replace(download_file,'.zip','')
    # and download it
    urllib.urlretrieve(amalgamation_url, "tmp.zip")
```

Once I applied such a trivial patch anything ran in the smoothest and most pleasant way.

Python sample program

spatialite_sample.py

```
# importing pyspatialite
from pyspatialite import dbapi2 as db

# creating/connecting the test_db
conn = db.connect('test_db.sqlite')

# creating a Cursor
cur = conn.cursor()

# testing library versions
rs = cur.execute('SELECT sqlite_version(), spatialite_version()')
for row in rs:
    msg = "> SQLite v%s Spatialite v%s" % (row[0], row[1])
    print msg

# initializing Spatial MetaData
# using v.2.4.0 this will automatically create
# GEOMETRY_COLUMNS and SPATIAL_REF_SYS
sql = 'SELECT InitSpatialMetadata()'
cur.execute(sql)

# creating a POINT table
sql = 'CREATE TABLE test_pt ('
sql += 'id INTEGER NOT NULL PRIMARY KEY,'
sql += 'name TEXT NOT NULL)'
cur.execute(sql)
# creating a POINT Geometry column
sql = "SELECT AddGeometryColumn('test_pt',
" sql += "'geom', 4326, 'POINT', 'XY')"
cur.execute(sql)

# creating a LINESTRING table
sql = 'CREATE TABLE test_ln ('
sql += 'id INTEGER NOT NULL PRIMARY KEY,'
sql += 'name TEXT NOT NULL)'
cur.execute(sql)
# creating a LINESTRING Geometry column
sql = "SELECT AddGeometryColumn('test_ln', "
sql += "'geom', 4326, 'LINESTRING', 'XY')"
cur.execute(sql)

# creating a POLYGON table
sql = 'CREATE TABLE test_pg ('
sql += 'id INTEGER NOT NULL PRIMARY KEY,'
sql += 'name TEXT NOT NULL)'
cur.execute(sql)
# creating a POLYGON Geometry column
sql = "SELECT AddGeometryColumn('test_pg', "
sql += "'geom', 4326, 'POLYGON', 'XY')"
cur.execute(sql)

# inserting some POINTs
# please note well: SQLite is ACID and Transactional
# so (to get best performance) the whole insert cycle
# will be handled as a single TRANSACTION
for i in range(100000):
    name = "test POINT #%d" % (i+1)
    geom = "GeomFromText('POINT("
    geom += "%f " % (i / 1000.0)
    geom += "%f" % (i / 1000.0)
    geom += ")', 4326)"
    sql = "INSERT INTO test_pt (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking POINTs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_pt"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

# inserting some LINESTRINGs
for i in range(100000):
    name = "test LINESTRING #%d" % (i+1)
    geom = "GeomFromText('LINESTRING("
    if (i%2) == 1:
    # odd row: five points
        geom += "-180.0 -90.0, "
        geom += "%f " % (-10.0 - (i / 1000.0))
        geom += "%f, " % (-10.0 - (i / 1000.0))
        geom += "%f " % (10.0 + (i / 1000.0))
        geom += "%f" % (10.0 + (i / 1000.0))
        geom += ", 180.0 90.0"
    else:
    # even row: two points
        geom += "%f " % (-10.0 - (i / 1000.0))
        geom += "%f, " % (-10.0 - (i / 1000.0))
```

```
        geom += "%f " % (10.0 + (i / 1000.0))
        geom += "%f" % (10.0 + (i / 1000.0))
    geom += ")', 4326)"
    sql = "INSERT INTO test_ln (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking LINESTRINGs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_ln"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

# inserting some POLYGONs
for i in range(100000):
    name = "test POLYGON #%d" % (i+1)
    geom = "GeomFromText('POLYGON(("
    geom += "%f " % (-10.0 - (i / 1000.0))
    geom += "%f, " % (-10.0 - (i / 1000.0))
    geom += "%f " % (10.0 + (i / 1000.0))
    geom += "%f, " % (-10.0 - (i / 1000.0))
    geom += "%f " % (10.0 + (i / 1000.0))
    geom += "%f, " % (10.0 + (i / 1000.0))
    geom += "%f " % (-10.0 - (i / 1000.0))
    geom += "%f, " % (10.0 + (i / 1000.0))
    geom += "%f " % (-10.0 - (i / 1000.0))
    geom += "%f" % (-10.0 - (i / 1000.0))
    geom += "))', 4326)"
    sql = "INSERT INTO test_pg (id, name, geom) "
    sql += "VALUES (%d, '%s', %s)" % (i+1, name, geom)
    cur.execute(sql)
conn.commit()

# checking POLYGONs
sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), "
sql += "ST_Srid(geom) FROM test_pg"
rs = cur.execute(sql)
for row in rs:
    msg = "> Inserted %d entities of type " % (row[0])
    msg += "%s SRID=%d" % (row[1], row[2])
    print msg

rs.close()
conn.close()
quit()
```

```
$ python spatialite_sample.py
```

Telling the full story is boring and very few interesting: you can study and test the sample code by yourself, and that's absolutely all.

Language bindings: PHP

2011 January 28

Test environment

Linux Debian:
just to be sure to check an up-to-date state-of-the-art I've actually used **Debian Squeeze** (32 bit).
So I'm actually sure that all required packages are reasonably using the most recent version.

PHP and SQLite connector:
My Debian Virtual Machine had no Apache and PHP stuff already installed.
So I started my test installing the following packages:

- **apache2**
- **php5-cli**
- **php5-sqlite**

Caveat

The most recent **PHP 5.3** seems to be absolutely required in order to support SpatiaLite.
I've performed some further test on the oldest **Debian Lenny**, simply to immediately discover that PHP (and sqlite) where so obsolete that using SpatiaLite was completely impossible.

I suppose that if you are strongly interested into using SpatiaLite updating to **PHP 5.3** is absolutely required before attempting any preliminary test.

Configuring PHP

I quickly discovered that using the default PHP configuration SpatiaLite cannot be dynamically loaded as an extension to the basic SQLite connector.
At least the following change has to applied first into the **`/etc/php5/apache2/php.ini`** configuration script.

default **`php.ini`**:

```
[sqlite3]
;sqlite3.extension_dir =
```

updated **`php.ini`**:

```
[sqlite3]
sqlite3.extension_dir = /var/www/sqlite3_ext
```

The SQLite connector for PHP actually has a built-in capability to load dynamic extensions.
Anyway you must explicitly enable a given directory containing any extension to be dynamically loaded.

```
# /etc/init.d/apache2 restart
```

After modifying the **`php.ini`** script restarting the Apache WEB server is absolutely required, so to materialize the

new configuration.

```
# mkdir /var/www/sqlite3_ext
# cp /usr/local/lib/libspatialite.so /var/www/sqlite3_ext
```

Then I've simply created the **`/var/www/sqlite3_ext`** directory.
And finally I've copied the **`libspatialite.so`** shared library form **`/usr/local/lib`** into this directory.

Please note well: in order to perform all the above mentioned operations you must login as **`root`**

PHP sample program

SpatialiteSample.php

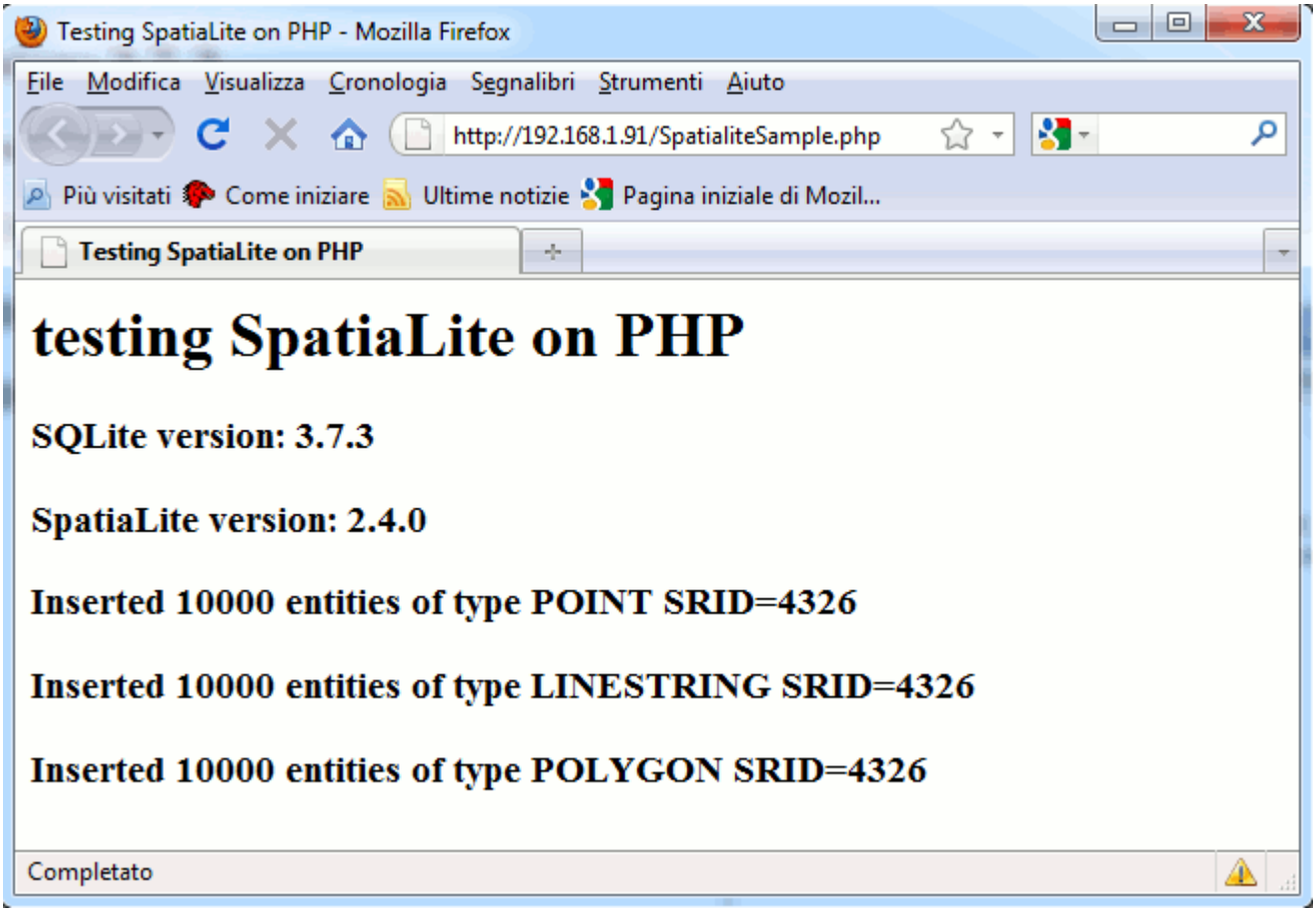
```
<html>
  <head>
    <title>Testing SpatiaLite on PHP</title>
  </head>
  <body>
    <h1>testing SpatiaLite on PHP</h1>

<?php
# connecting some SQLite DB
# we'll actually use an IN-MEMORY DB
# so to avoid any further complexity;
# an IN-MEMORY DB simply is a temp-DB
$db = new SQLite3(':memory:');

# loading SpatiaLite as an extension
$db->loadExtension('libspatialite.so');

# enabling Spatial Metadata
# using v.2.4.0 this automatically initializes SPATIAL_REF_SYS
# and GEOMETRY_COLUMNS
$db->exec("SELECT InitSpatialMetadata()");

# reporting some version info
$rs = $db->query('SELECT sqlite_version()');
while ($row = $rs->fetchArray())
{
  print "<h3>SQLite version: $row[0]</h3>";
}
$rs = $db->query('SELECT spatialite_version()');
while ($row = $rs->fetchArray())
{
  print "<h3>SpatiaLite version: $row[0]</h3>";
}

# creating a POINT table
$sql = "CREATE TABLE test_pt (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);
# creating a POINT Geometry column
$sql = "SELECT AddGeometryColumn('test_pt', ";
$sql .= "'geom', 4326, 'POINT', 'XY')";
$db->exec($sql);

# creating a LINESTRING table
$sql = "CREATE TABLE test_ln (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);
# creating a LINESTRING Geometry column
$sql = "SELECT AddGeometryColumn('test_ln', ";
$sql .= "'geom', 4326, 'LINESTRING', 'XY')";
$db->exec($sql);

# creating a POLYGON table
$sql = "CREATE TABLE test_pg (";
$sql .= "id INTEGER NOT NULL PRIMARY KEY,";
$sql .= "name TEXT NOT NULL)";
$db->exec($sql);
# creating a POLYGON Geometry column
$sql = "SELECT AddGeometryColumn('test_pg', ";
$sql .= "'geom', 4326, 'POLYGON', 'XY')";
$db->exec($sql);

# inserting some POINTs
```

```
# please note well: SQLite is ACID and Transactional
# so (to get best performance) the whole insert cycle
# will be handled as a single TRANSACTION
$db->exec("BEGIN");
for ($i = 0; $i < 10000; $i++)
{
  # for POINTs we'll use full text sql statements
  $sql = "INSERT INTO test_pt (id, name, geom) VALUES (";
  $sql .= $i + 1;
  $sql .= ", 'test POINT #";
  $sql .= $i + 1;
  $sql .= "', GeomFromText('POINT(";
  $sql .= $i / 1000.0;
  $sql .= " ";
  $sql .= $i / 1000.0;
  $sql .= ")', 4326))";
  $db->exec($sql);
}
$db->exec("COMMIT");

# checking POINTs
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
$sql .= "ST_Srid(geom) FROM test_pt";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
  # read the result set
  $msg = "Inserted ";
  $msg .= $row[0];
  $msg .= " entities of type ";
  $msg .= $row[1];
  $msg .= " SRID=";
  $msg .= $row[2];
  print "<h3>$msg</h3>";
}

# inserting some LINESTRINGs
# this time we'll use a Prepared Statement
$sql = "INSERT INTO test_ln (id, name, geom) ";
$sql .= "VALUES (?, ?, GeomFromText(?, 4326))";
$stmt = $db->prepare($sql);
$db->exec("BEGIN");
for ($i = 0; $i < 10000; $i++)
{
  # setting up values / binding
  $name = "test LINESTRING #";
  $name .= $i + 1;
  $geom = "LINESTRING(";
  if (($i%2) == 1)
  {
    # odd row: five points
    $geom .= "-180.0 -90.0, ";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= " ";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= ", ";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= " ";
    $geom .= 10.0 + ($i / 1000.0);
    $geom .= ", ";
    $geom .= 10.0 + ($i / 1000.0);
    $geom .= " ";
    $geom .= 10.0 + ($i / 1000.0);
    $geom .= ", 180.0 90.0";
  }
  else
  {
    # even row: two points
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= " ";
    $geom .= -10.0 - ($i / 1000.0);
    $geom .= ", ";
    $geom .= 10.0 + ($i / 1000.0);
    $geom .= " ";
    $geom .= 10.0 + ($i / 1000.0);
  }
  $geom .= ")";

  $stmt->reset();
  $stmt->clear();
  $stmt->bindValue(1, $i+1, SQLITE3_INTEGER);
  $stmt->bindValue(2, $name, SQLITE3_TEXT);
  $stmt->bindValue(3, $geom, SQLITE3_TEXT);
  $stmt->execute();
}
$db->exec("COMMIT");
```

```
# checking LINESTRINGs
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
$sql .= "ST_Srid(geom) FROM test_ln";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
  # read the result set
  $msg = "Inserted ";
  $msg .= $row[0];
  $msg .= " entities of type ";
  $msg .= $row[1];
  $msg .= " SRID=";
  $msg .= $row[2];
  print "<h3>$msg</h3>";
}

# insering some POLYGONs
# this time too we'll use a Prepared Statement
$sql = "INSERT INTO test_pg (id, name, geom) ";
$sql .= "VALUES (?, ?, GeomFromText(?, 4326))";
$stmt = $db->prepare($sql);
$db->exec("BEGIN");
for ($i = 0; $i < 10000; $i++)
{
  # setting up values / binding
  $name = "test POLYGON #";
  $name .= $i + 1;
  $geom = "POLYGON((";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= " ";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= ", ";
  $geom .= 10.0 + ($i / 1000.0);
  $geom .= " ";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= ", ";
  $geom .= 10.0 + ($i / 1000.0);
  $geom .= " ";
  $geom .= 10.0 + ($i / 1000.0);
  $geom .= ", ";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= " ";
  $geom .= 10.0 + ($i / 1000.0);
  $geom .= ", ";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= " ";
  $geom .= -10.0 - ($i / 1000.0);
  $geom .= "))";

  $stmt->reset();
  $stmt->clear();
  $stmt->bindValue(1, $i+1, SQLITE3_INTEGER);
  $stmt->bindValue(2, $name, SQLITE3_TEXT);
  $stmt->bindValue(3, $geom, SQLITE3_TEXT);
  $stmt->execute();
}
$db->exec("COMMIT");

# checking POLYGONs
$sql = "SELECT DISTINCT Count(*), ST_GeometryType(geom), ";
$sql .= "ST_Srid(geom) FROM test_pg";
$rs = $db->query($sql);
while ($row = $rs->fetchArray())
{
  # read the result set
  $msg = "Inserted ";
  $msg .= $row[0];
  $msg .= " entities of type ";
  $msg .= $row[1];
  $msg .= " SRID=";
  $msg .= $row[2];
  print "<h3>$msg</h3>";
}

# closing the DB connection
$db->close();
?>

  </body>
</html>
```

I saved this PHP sample script as: **/var/www/SpatialiteSample.php**

Then I simply started my Firefox WEB browser requesting the corresponding URL:



And that's all.

Please note: may well be that using other different Linux distros (or Windows) adjusting any *pathname* as appropriate for your specific platform should be required.

Author: Alessandro Furieri a.furieri@lqt.it

This work is licensed under the Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license.

Permission is granted to copy, distribute and/or modify this document under the terms of the
GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.